A GRAPHICS ENVIRONMENT SUPPORTING THE RAPID PROTOTYPING OF PICTORIAL COCKPIT DISPLAYS

THESIS

Alan J. Braaten
Captain, USAF

AFIT/GCS/MA/86D-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

A GRAPHICS ENVIRONMENT SUPPORTING THE RAPID PROTOTYPING OF PICTORIAL COCKPIT DISPLAYS

THESIS

Alan J. Braaten
Captain, USAF

AFIT/GCS/MA/86D-1

AFIT/GCS/MA/86D-1

A GRAPHICS ENVIRONMENT SUPPORTING THE RAPID PROTOTYPING OF PICTORIAL COCKPIT DISPLAYS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Masters of Science in Information Science

Alan J. Braaten, B.S.
Captain, USAF

December 1986

Acknowledgments

I would like to thank my advisor, Ms. Karyl Adams, for her support in this effort. It was her initial interests and enthusiam that lead me to pursue this research. I would also like to thank fellow students, Captain Markoe Hanson and Captain Michael Denny, for their stimulating and often enlightening discussions throughout this thesis endeavor. And finally, I owe my warmest appreciation to my wife, Karen, and our three children, Jesse, Josie, and Evan for their loving patience and support.

Alan J. Braaten

Table of Contents

List of Figures

List of Tables

List of Acronyms

AC/CSRL	Advanced Cockpit/Crew Station Research Laboratory
AFIT	Air Force Institute of Technology
AFWAL	Air Force Wright Aeronautical Laboratory
ALC	Automated Layout Center
DESIGNS	Display Environment Supporting the Interactive Generation of alphaNumerics and Symbology
HUD	Heads Up Display
OOD	Object-Oriented Design

Abstract

The purpose of this investigation was to design and implement a graphics based environment capable of supporting the rapid prototyping of pictorial cockpit displays. Attention was focused on the interactive construction of pictorial type cockpit displays from libraries of cockpit displays and symbology.

Implementation was based on an object-oriented programming paradigm. This approach provided a natural and consistent means of mapping abstract design specifications into functional software. Implementation was supported by an object-oriented extension to the 'C' programming language.

Although this investigation addressed a specific application, the resulting graphic environment is applicable to other areas requiring the rapid prototyping of pictorial displays.

A GRAPHICS ENVIRONMENT SUPPORTING THE RAPID PROTOTYPING OF PICTORIAL COCKPIT DISPLAYS

1. Introduction

The demand for cockpit displays has currently surpassed the capabilities for generating them. An ever-increasing time lag between requirements definition and implementation has developed. Cockpit display generation can take from weeks to years to implement adding to development cost and reducing research effectiveness [AFWAL AC/CSRL Technical Program Plan, 1985]. Two major factors have contributed to the current cockpit display shortage. They are the lack of automated design tools and the lack of a reusable software base for cockpit display development.

Currently, cockpit display design is a tedious, manual process. Tools to support rapid prototyping of generic cockpit displays are almost non-existant. The few tools that do exist are usually tailored to a specific display type (e.g. Head Up Displays) [Adams, 1985] severely restricting their applicability to other display types. Software bases suffer from similar limitiations. They either don't exist, must be generated manually, are tailored to a specific cockpit display type, or are so device dependent that portability is impossible.

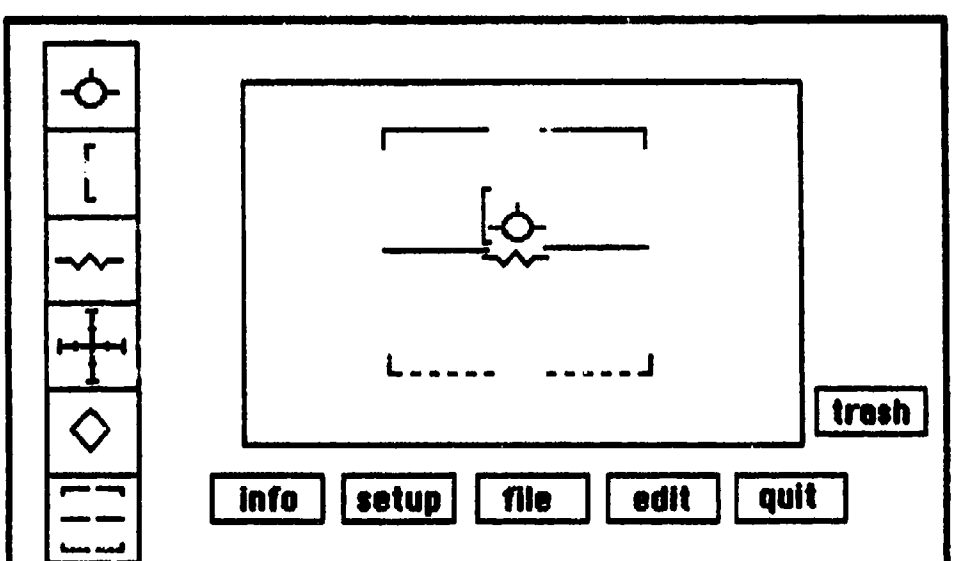

Figure 1. Heads Up Display.

This thesis proposed a partial solution to this problem by:

1) developing a set of automated design tools capable of supporting the rapid prototyping of multiple cockpit display types, and

2) establishing a reusable on-line software base to aid cockpit display construction.

These solutions were realized in a highly integrated, interactive graphics environment. This environment provided a dynamic medium for the rapid prototyping of multiple display types by providing the capability of constructing cockpit layouts and cockpit symbology from an on-line software base of cockpit layouts and cockpit symbology. Layouts were constructed, in a building block fashion, by selecting existing cockpit layouts and/or cockpit symbology from a software base and pasting them on a representation of an actual cockpit display. Figure 1 illustrates a Heads Up Display (HUD) constructed via this methodology. Individual cockpit symbols are constructed in a similar manner from a set of graphic

primitives. The ability to construct both cockpit layouts and cockpit symbology on-line makes this environment a feasible solution for the cockpit design/research problem.

Background

Current Methodology. Currently, the design and implementation of a cockpit layout design is a long and tedious process. It requires many steps, each performed manually by different experts. These steps are:

- Design Layout and Analysis
- Design Translation to Software
- Integration with Aircraft Simulation Models
- Testing and Evaluation

Design layout and analysis is the act of capturing the designer's ideas on paper. Cockpit layouts are assembled in a building block fashion from pre-defined sets of cockpit components. Components are selected and positioned on the layout. New components are created and saved as requirements dictate. The final layout can be viewed as a collection of cockpit components spatially arranged to satisfy a design requirement.

When the layout is complete, it is transferred to software and simulation experts for translation into computer code. Each component may be realized with an associated subroutine or procedure. It is the responsibility of the software and simulation experts to create source code files, composed of the various procedures, into a single application for testing. New components, with no associated code, will require individual design, coding, and testing before they are integrated into the final application.

The amount of time and code required during this phase depends directly on the complexity of the layout. Layouts that are slight modifications to existing ones, require a minimal amount of software to be generated. A new layout with new components will require extensive time and software resources. In either case, design translation is an expensive and time consuming activity. It requires extensive debugging and testing of each individual component and system testing of the final design.

After the design has been translated to code, it is integrated with various aircraft simulation models for testing. This step can also be very time consuming and resource expensive. The result of this step is a complete software simulation of the cockpit design constrained by the specific aircraft characteristics, ready for testing by the designer and the intended user.

The final software product is then installed within a simulation cockpit for testing. It is at this stage that the user and designer can actually 'fly' the design. If testing reveals that modifications to the current layout are necessary, the layout is re-cycled through the process outlined above. Although redesign is usually not as an extensive development effort as the original design, it still consumes a considerable amount of time and support resources.

Problems. Four major problems exist with current methodology; they are as follows:

- Tedious (manual) Process
- Time Consuming
- Non-Responsive
- Device Dependent

One of the main problems with the current methodology is that virtually all steps are performed manually. Design layout, analysis, and any required corrections are manually performed on paper. Design translation is performed by software and simulation experts who manually write, debug, and test the code. The resulting code is then manually integrated with aircraft simulation models for dynamic testing.

Not only is the current design process tedious, it is also time consuming. Depending on the complexity of the design, implementation (from design conception to testing and evaluation) can take from weeks to years to complete. By the time the design is implemented, the designer is pursuing other layout schemes or the current layout is no longer pertinent.

The drudgery of the manual process, coupled with an ever-increasing time delay between idea conception and implementation, fosters a design process that is non-responsive to the designer's needs. The current process does not provide the designer with an effecive or efficient means of evaluating and modifying the design in a continuous, and timely, fashion. The designer becomes as outside participant in the design process once the design translation begins. Not until the test and evaluation phase can the designer interject inputs to the design process. Then, all errors or modifications identified result in the software being turned over to software and simulation experts for re-work. This can entail lengthy modification, testing, and debugging before the software is returned to the designer. Simple changes can take weeks to implement. The ability to fine tune designs becomes almost impossible due to the time and resource expenses involved.

Another factor contributing to the non-responsive, inflexible nature of the current design process is the software dependencies on specific hardware devices. Due to the heterogeneous nature of the hardware environment, much of the software written to implement cockpit displays is device dependent. Rehosting a design (i.e. the software) often entails a duplicate development effort. The original software may require a complete re-design to function on the new hardware. Even a minimal rehosting effort will require extensive testing and debugging of the new software in its new hardware environment to ensure compliance with original requirements.

DESIGNS. Research to provide an alternative to the current methodology was undertaken and reported in the 1985 Air Force Institute of Technology (AFIT) thesis entitled "A Display Environment Supporting the Interactive Generation of alphaNumerics and Symbology with DESIGNS on the Future" (referred to as DESIGNS). DESIGNS had two goals: 1) to demonstrate the feasibility of using a graphics based environment for the generation and editing of display formats and 2) the automatic generation of source code from the display format for a targeted graphics device [Adams, 1985:50].

To accomplish these goals, an initial system was implemented to develop and modify HUD formats. The designer could interactively lay out HUDs from pre-defined sets of HUD symbols. Code was automatically generated from a layout by linking together source code associated with each HUD symbol in the display. The source code and the symbols were maintained in on-line libraries. Manual intervention was confined to the layout process.

DESIGNS significantly reduced the time required for static HUD implementation from weeks/months to an average of 30 minutes. The 30

minutes included actual HUD layout activity and subsequent software generation [Adams, 1986]. It should be emphasized that all HUD layouts were constructed from pre-existing symbol sets. Dynamic inclusion of new symbols was not directly supported by DESIGNS. The designer was still dependent on software experts to manually integrate new symbols into the DESIGNS environment.

AC/CSRL. Using DESIGNS as a baseline for development, the Air Force Wright Aeronautical Laboratories, Flight Dynamic Laboratory (AFWAL/FIGR, FIGD) initiated research directed toward implementing an Advanced Cockpit/Crew Station Research Laboratory (AC/CSRL). The goal of the AC/CSRL is to automate the entire cockpit design process, with the main objective of keeping the designer as the focal point of all aspects of the design process. Currently the AC/CSRL is in definition phase; implementation is not expected until the 1990's.

AC/CSRL is expected to support the following:

- interactive design and modification of cockpit layouts,
- automatic generation of source code from the design,
- automatic linkage to aircraft simulation models, and
- one day turnaround time to prepare completely new cockpit arrangements for testing [AFWAL AC/CSRL Technical Program Plan, 1985].

Unlike DESIGNS, which only supported the development of HUD-type displays, the AC/CSRL will support a variety of displays, from simple alphanumerics to complex pictorial type displays. In addition, the AC/CSRL will also support dynamic creation and integration of new cockpit symbology into the design environment.

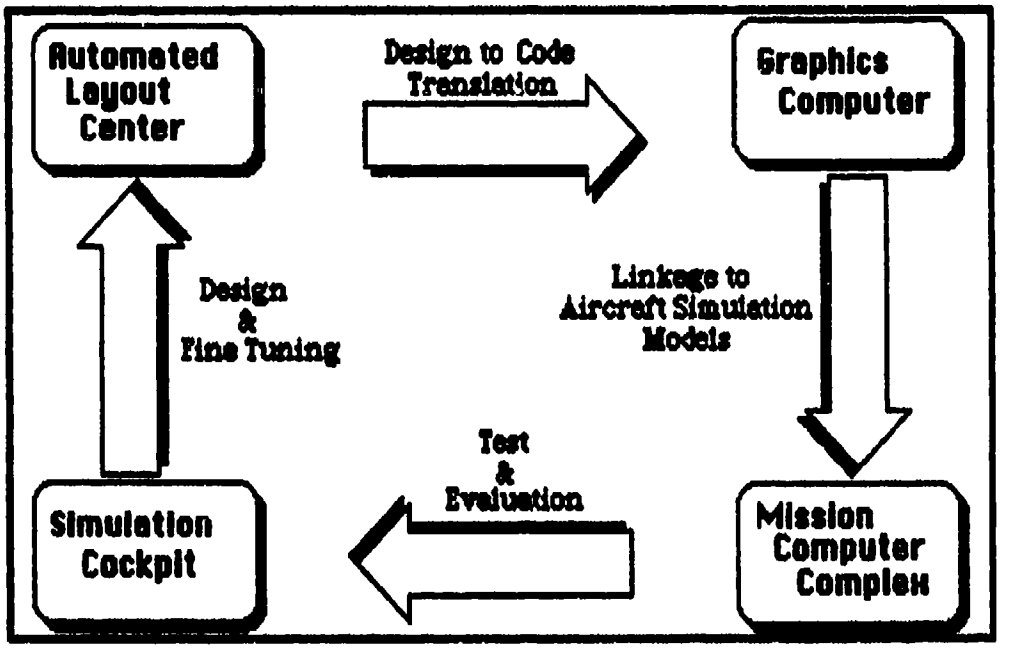

Figure 2. AC/CSRL Concept.

To accomplish this, the AC/CSRL will rely heavily on a flexible graphical interface and on on-line libraries of cockpit symbology and associated source code. Displays will be constructed in a building block fashion, by selecting cockpit symbols from the libraries and placing them on a repesentation of an actual cockpit display. Source code will then be automatically generated and combined with aircraft simulation packages to produce a full mission scenario. The designer, in effect, will be able to construct a design layout, generate the corresponding code, and dynamically test the design within the same environment. The design process will no longer be dependent upon software and simulation experts, thus improving turnaround time and reducing costs. Figure 2 conceptually depicts the AC/CSRL concept with its associated components.

At a procedural level the AC/CSRL will not drastically alter the nature of the design process. The same procedures performed in the current manual design process will also be performed via the AC/CSRL. However, the

degree of manual intervention is significantly different between the two. The current process is almost entirely manual. Few, if any, of the steps are automated. In contrast, the goal of the AC/CSRL is to automate the entire process.

Design layout and analysis will be supported interactively via a flexible graphics interface, known as the Automated Layout Center (ALC). The ALC is the focal point of all cockpit design activity. All cockpit layout design as well as cockpit symbology construction is handled through the ALC. It provides the designer with a window into the AC/CSRL environment.

Testing will be significantly enhanced. Complete simulation senarios will be generated for the designer and intended user to test (i.e. 'fly') and evaluate. Redesign will be supported in a more timely manner. It should be possible to incorporate modifications and retest a redesign within the same day, as opposed to weeks in the current process. This should significantly improve productivity and promote experimentation of alternative display representations.

Problem Statement

The goal of this thesis research was to demonstrate, in part, the feasibility of the AC/CSRL concept by designing and implementing a prototype of the Automated Layout Center. The prototype ALC focused on the rapid prototyping of pictorial type cockpit displays via the use of on-line libraries of cockpit layouts and cockpit symbology. In addition to addressing a rapid prototyping capability, this thesis also presented a candidate user interface for the ALC.

The implementation of the prototype ALC was based on an object-oriented paradigm. An object-oriented approach was chosen because it provided a framework that was a "direct and natural correspondence between the world (*i.e. the design process*) and its model (*i.e. a virtual cockpit display*)" [Borgida et al., 1985:85] (italic phrases added by author). To suppport such an implementation, an object-oriented extension of the 'C' programming language was implemented. These extensions were modeled after the Smalltalk environment [Goldberg and Robson, 1983].

Scope

The objective of this thesis was to design and implement a prototype ALC supporting the rapid prototyping of cockpit displays. The results from this effort were to be used as guidelines for future AC/CSRL related projects. Since this was a rather ambitious project, the following constraints were placed on this research effort.

Breath of Implementation. The prototype ALC design consisted of four functional areas; 1) the user interface, 2) the Layout Editor, 3) the Symbol Editor, and 4) the Librarian. The user interface provided the dynamic medium through which all user interactions and system responses were handled. The Layout Editor, a graphics editor, supported the dynamic prototyping of cockpit layouts. The Symbol Editor, another graphics editor, supported the creation and modification of cockpit symbology. The Librarian provided an archival mechanism used to store and retrieve cockpit layouts and cockpit symbols.

Due to time constraints and for demonstation purpose, only the user interface and Layout Editor were implemented. The exclusion of the Symbol Editor and Librarian did not significantly distract from the goal of demonstrating the feasibility of the ALC concept. However, for completeness and future implementation, the requirements definition and design of all four functional areas are presented in this thesis.

Display Dimensionality. The AC/CSRL is expected to support the creation of both two and three dimensional cockpit display representations. The prototype ALC supported only the design and representation of two dimensional cockpit displays. Three dimensional cockpit display representation will be an essential component of future displays; however, including this capability within this thesis would only serve to distract from the fundamental characteristics of the ALC that needed to be addressed first. The ALC prototype design does not explicitly rule out the inclusion of three dimensional capabilities, but on the other hand, it does not explicitly address it either.

Code Generation. The AC/CSRL will automatically generate source code from the cockpit designs created on the ALC. Code generation is not addressed in this thesis. The prototype ALC only supports the creation and modification of cockpit layout formats; it does not generate source code from said layouts.

Input Devices. This thesis cannot address the merits of which input device(s) should be used to interface with the prototype ALC. Although the AC/CSRL will support many devices, time constraints and the lack of

available input devices for interfacing prohibit consideration in this thesis. As such, the primary means of interaction will be via a mouse. The keyboard is used for retrieval of textual information only (i.e. filenames, labels, etc.).

Transportability. A key requirement of the AC/CSRL is device independence. Device independence will allow re-targeting of the AC/CSRL designs on different graphic devices. This is not attainable for the prototype ALC due to the unavailability of appropriate systems.

Although the design is device independent, the implementation is targeted to a Raster Technologies Model One/25 graphics system. As such, some sections of the code will be device dependent. These sections have been identified and isolated as much as possible.

Sequence of Presentation

The second and third chapters of this thesis present the system requirements and overall system design. Chapter 2 addresses system requirements related to three general areas; hardware, software, and the user interface for the prototype ALC. Chapter 3 maps the system requirements into four functional areas that comprise the ALC prototype's design, namely; the user interface, Layout Editor, Object Editor, and the Librarian.

The fourth, fifth, and sixth chapters describe in detail the design of the four functional area defined in Chapter 3. Chapter 4 addresses the design of the user interface, discussing what design considerations were followed and the actual interactive components that comprise the user interface. Chapter 5 discusses the similarities and differences between the Layout

Editor and the Symbol Editor. Chapter 6 provides insight into the design of the prototype ALCs archival mechanism, the Librarian.

Chapter 7 deals with the actual implementation of the ALC prototype. Implementation is approach from three viewpoints. The first view is a description of the actual host hardware for the ALC prototype. This is followed by a description of the object-oriented environment used in implementing the ALC prototype software. And finally, a description of the ALC prototype implementation is presented. Chapter 8 concludes the written thesis with conclusions regarding the success of this research effort and recommendations for the future.

The appendices of this thesis provide information relative to the software and hardware environments in which the prototype ALC was hosted. Specifically, Appendix A addresses the design approach followed in the conceptualization of the software. This is followed in Appendix B with a basic overview of the object-oriented concepts that were used to model the implementation of the prototype ALC software. Appendix C provides a detailed discussion of the object-oriented extensions to the 'C' language that were implemented to support the prototype ALC implementation. Appendix D outlines the graphic capabilities (i.e. classes) that are currently supported by the object-oriented implementation. Appendices E and F describe the operational characteristics of the Raster Technologies Model One/25 and associated device drivers.

II. Requirements

There are three primary areas of requirements for the ALC prototype that this chapter will define. They are the hardware, software, and the user interface areas. Hardware requirements encompass the actual hardware needed to support the prototype ALC. The software requirements section describes the functional requirements and provides guidelines for software development. User interface requirements bind the hardware and software requirements together. A detailed description of each requirement category is presented in the following sections.

Hardware

Although no hardware is being designed, and the target host has already been identified, it is still worth mentioning, in general terms, the hardware requirements needed to support the ALC prototype. This section provides general guidelines to follow if the ALC prototype is re-hosted at some future time.

There are four general areas that the hardware discussion should address. These areas are display technology, input devices, output devices, and processing and storage capabilities [Rose, 1982:25].

Display Technology. Three basic types of display technologies could be used for the ALC prototype: vector, storage tube, and raster. Each technology uses a technique known as 'phosphorescence' to illuminate an image on the display screen. This process involves the use of an electron

beam to excite a phosphor-coated display screen. The phosphor when excited, jumps to a higher, unstable energy state. When the beam is removed, the phosphor returns to its original stable state releasing the excess energy as light. Images are drawn by directing an electron beam on the phosphor-coated screen in the desired shape or pattern. The method used to direct the electron beam constitutes the major distinction between the three technologies.

Vector (or often refered to as stroke, or calligraphic) displays display images by directly tracing the image on the display screen with the electron beam. This method is extemely fast and straightforward. However, because the illuminated phosphor fades at a exponential rate, the image must be continuously retraced (or refreshed) for the image to remain on the screen [Foley and Van Dam,1982:106]. As more and more images are drawn, an annoying flicker in the display presentation may develop.

Storage tube displays circumvent the flicker problem by tracing the image on a fine mesh grid mounted immediately behind the phosphor coated screen. Images traced on the grid are transferred directly on to the phosphor. The grid acts as a storage medium, saving the image once it is traced. This allows the image to be drawn only once, eliminating the flicker problem caused by the need to refresh the screen constantly. However, since the image is stored on the grid and continuously displayed, it becomes almost impossible to make selective erasures from the screen. In order to erase a single image, the entire display must be erased and the modified image redrawn.

Raster displays differ fundamentally in the way images are drawn. Unlike the refresh and storage tube displays, the image is not directly traced on the screen. Rather it is written into a storage area know as a

'frame buffer'. A frame buffer is a matrix of bits, each corresponding to a unique address point (or pixel) on the screen. Each entry in the matrix stores the brightness and/or color value for its corresponding screen pixel. An image is displayed by processing the matrix, row by row, using the contents of each entry to control the electron beam intensity. The image is thus displayed row by row, starting at the top of the screen and finishing at the bottom.

Raster displays overcome the problems of screen flicker and erasure problems associated with vector and storage tube displays. The electron beam is not required to bounce around the screen tracing an image but rather follows a predefined pattern (row by row) with in a predefined time span (30 to 60 times a second). This allows a display composed of many images to be drawn at the same rate as a display containing just a few images. The frame buffer provides a means of doing selective erasures without the entire screen having to be erased and redrawn.

The choice of which technology to use often depends on the intended application. For applications that require only static displays and minimal user interaction; all three technologies could be used. For dynamic, user intensive application, such as the ALC prototype, a vector or raster display would be warranted. Table I provides a summary of the capabilities of the three display technologies [Dudley, 1982:38].

Input Devices. The ALC prototype requires, as a minimum, an input device capable of performing the following two functions, 1) cursor tracking, and 2) object selection. A number of input devices could satisfy this requirement. The most commonly used devices are, the tablet or digitizer, touchpanel, joystick, mouse, and trackball. The choice of which

TABLE I

Display Technology Summary

	Raster	Storage Tube	Vector
Resolution	low to high	high	very high
Drawing type	areas, realistic	line	line
Motion	dynamic	static	dynamic
Color	millions	green on green	mono or 4 to 8 colors
Interactiveness	high	low	very high
Use	intelligent terminal, stand-alone	dumb terminal	stand-alone system

device is used will most likely depend on availability of the input device. Table II summarizes software development cost factors, input type, and special considerations related to these devices [Ohlson, 1979:285].

Output Devices. The ALC prototype does not require an output device other than the display screen. If hardcopies are desired, a plotter or film recorder could be used.

Processing and Storage. The central processing unit (CPU) is the heart of a graphics system. Regardless of whether the graphics system is a stand-alone or remote terminal, the computational processing capabilites of the CPU will directly influence the type of graphic applications that can be implemented and expected to execute within a reasonable amount of time.

TABLE II

Input Device Summary

Device	Software Dev. Cost	Input Type	Special Considerations
Tablet	Medium	Indirect Graphical	Some Units are not suitable for online interactive use.
Touchpanel	Low	Direct Tactile	Gross resolution, makes detail work impossible.
Joystick	Low	Indirect Tactile	Large variety, fits most applications.
Mouse	Low	Indirect Tactile	Relative Positioning.
Trackball	Low	Indirect Tactile	Slewing capabilities.

Computationally intensive applications such as 3-dimensional modeling and ray tracing require significant processing power, as opposed to simple line chart applications. The choice of which CPU to use should be weighed against the computational aspect of the application. The ALC prototype does not require an exceptionally powerful CPU. Most 16-bit microprocessors available on the market today would suffice.

Storage, both internal (memory) and external (secondary), also influence the speed at which an application can execute. Applications constrained by limited main memory or hampered by slow peripherals, often spend a significant amount of time waiting for segments of the application to be swapped to and from memory or waiting on data transmission from the

TABLE III

Prototype ALC's Hardware Requirements.

Requirement	Implementation
Display Technolgy	Refresh or Raster
Input Device	Cursor Tracking and Object Selection
Output Device	Optional
Processing and Storage	16 bit micro (minimum) 1M main memory, 10M disk

peripheral device. Instead of performing useful work, the application becomes 'I/O bound'. To prevent this from occuring with the ALC prototype, the host system should have at least 1 megabyte of main memory and, at a minimum, a 10 megabyte hard disk. Table III provides a summary of the ALC prototype hardware requirements.

Software

Software requirements for the ALC prototype can be classified into two areas, 1) software development requirements, and 2) implementation requirements. Software development requirements apply directly to the software development activity. They serve as guidelines to ensure that the software is developed in a consistent and standard way. They are general in

nature, and should be applied to all software development efforts regardless of the application. Implementation requirements, on the other hand, are application specific. They categorially state the functions that the software must perform. They address 'what' should be implemented, not 'how'.

Software Development Requirements. The principles listed here, are general software requirements that should be adhered to in the design and development of the actual software. These principles should be applied at all levels of the development effort.

Portability (device/host independence). The prototype ALC should be designed with host independence in mind. Since this is a prototype, it might be desirable to re-host this system on different workstations. As such, the software should be developed with no specific host in mind. Portions of the software that are dependent on the hardware should be isolated as much as possible and clearly identified.

Modularity. The software should be designed in a modular manner. Modularity provides a method of isolating the functions of the software into well-defined units. These units range in complexity from procedure to library packages [Fairley, 1985:145]. Some of the benefits modularity provides are: ease of implementation, maintenance, debugging, and complexity management [Shooman, 1983:110].

Simplicity. The software should be written so it is easy to read and understand. All software modules should contain only one entry and one exit point. They should conform to a consistent programming style and

should not attempt to hide the logic of the function under clever coding [Fairley, 1985:209-214]. It should be self-evident what a module does. Following such conventions makes the software easier to read, to understand, and to modify.

Efficiency. Efficiency refers to the ability of the software to operate under the current set of available resources. There are two main facets of efficiency: time and space [Booch, 1983:25]. Time efficiency pertains to the ability of the software and hardware to operate within a specified time constraint. The only time constraints imposed on this project are to provide a flicker-free display and response to user commands in a reasonable time. The system should wait on the user, not vice versa.

Space efficiency implies that the software should reside and execute within the current available memory. It should not be a requirement of this research to acquire additional hardware of any type.

Extensibility. To accommodate future modifications, the software should be designed with generality in mind. In other words, the software should provide a framework (harness) in which new capabilities (modules) can be 'plugged in' [Clemons and Greenfield, 1985:40]. This will allow the system to expand as user requirements change and advanced features are added.

Table IV summarizes of the software development requirements.

Implementation Requirements. Implementation requirements describe what the user wants the software to do. They are application specific and need to be clearly identified before the design or

TABLE IV

Software Development Requirements.

Requirement	
Portability	Host Independent
Modularity	Isolate the functions of the software into well defined units
Simplicity	Easy to read and learn
Efficiency	Operate with current set of available resources
Extensibility	Easy to add-on new capabilities

implementation begins. There are five basic functions that the ALC prototype must support:

- Graphical Interaction,
- Direct Manipulation,
- Iterative Development,
- Experimentation, and
- Evolutionary Design.

Graphical Interaction: The process of designing a cockpit display is a spatially oriented activity. It relies heavily on graphical images to represent cockpit and real-world objects. As such, the prototype ALC should exploit the use of graphics as a medium for the computer-human

interaction. The use of graphics, as a medium for interaction, provides certain distinct advantages over the use of text as a medium [Raeder, 1985:12] which are particularly relevant to the ALC prototype and related AC/CSRL efforts. These advantages are discussed in the following paragraphs.

The use of graphics as a medium of interaction permits instant random access (viewing) to any part of the display screen. The user can focus attention on a specific aspect of the display or can 'back-off' to grasp the overall structure. The user is not forced to follow a linear search pattern, but can switch from object to object, view to view, in a random fashion.

Text, on the other hand, forces the user to view information sequentially (usually top to bottom, left to right). It becomes difficult to grasp the information content of text displays. Headlines, bold type, and paragraphs provide some relief, but the process still remains sequential in nature.

Graphics provide multiple levels of dimensionality to the information being displayed. Information can be portrayed in two or three dimensions and the physical attributes of the information (i.e. shape, color, size, etc) can also be modified. Text, on the other hand, is a one-dimensional string of characters.

Graphic mediums also capitalize upon the inherent image processing capabilities of the human sensory system. The mind, a kind of biological image processor, is extremely adept at accessing and processing visual information. We tend to conceptualize things as pictures, not words. As the saying goes, 'A picture is worth a thousand words'.

The use of graphics as the main means of interaction directly influences the designer's perception of cockpit display creation. The process

of creating a cockpit display is very similar to computer programming. Instead of using a textual-based language such as Pascal or Ada to develop an application, the designer uses a graphics-based language to build (program) a display. The immediate details of syntax and control structure are transparent to the designer. The designer deals directly with the semantics of the language, understanding the whole verses individual parts or commands.

Direct Manipulation. An important aspect of graphical interaction is the ability to manipulate objects directly on the screen [Shneiderman, 1983:57]. Direct manipulation refers to the ability of the user to modify the characteristics of an object (i.e. shape, size, color, position, etc.) via some action. The user is not required to use an intermediate form (e.g. commands issued from the keyboard) to initiate an action. The user can use some form of a pointing device to select the object and then directly manipulate its form. Direct manipulation of cockpit objects is a key requirement of this system [AFWAL AC/CSRL Technical Program Plan, 1985].

Iterative Development. The process of developing cockpit displays is best implemented via an iterative process. An iterative process allows the user to evaluate and modify the design on a continuous basis [Shooman, 1983: 36]. This process should provide the flexibility to support the following required activities [AFWAL AC/CSRL Technical Program Plan, 1985]:

- implementation of new designs,
- implementation of design changes,
- the fine-tuning of designs

- creation and modification of cockpit symbology, and
- the rapid generation of alternative displays.

Experimentation. The system should be supportive of experimentation. It should allow the designer to experiment with possible design layouts without commitment. In other words, actions performed by the designer should be 'undo-able' [Harslem, 1984:104]. The ability to 'undo' a design decision allows the designer to explore alternative design representations without committing expensive and time consuming software and personnel resources.

A major problem plaguing the current design process is the inability to experiment with cockpit representations 'on-the-fly'. The designer is not afforded the luxury of making minor modifications to a cockpit layout without incurring additional time and resources to the already expensive design effort. This often leads to acceptance of the first design, simply because the costs associated with redesign (experimentation) are too prohibitive.

Evolutionary Design. "Contrary to the idea that a computer is exciting because a programmer can create something from seemingly nothing, our users were shown that a computer is exciting because it can be a vast storehouse of already existing ideas (models) that can be retrieved and modified for the user's personal needs. Programming should be viewed and enjoyed as an evolutionary rather than a revolutionay act" [Goldberg and Ross, 1981:354] The idea of an evolutionary design philosophy is very applicable and appealing for the design of cockpit displays. The designer creates the display in a building block fashion from sets of pre-existing

TABLE V

Prototype ALC Functional Software Requirements.

Requirement	
Graphical Interaction	Use of graphics vs. text
Direct Manipulation	Graphical objects directly manipulated by the user.
Interative Development	Evaluation and modification of the design on a continuous basis.
Experimentation	Design without commitment
Evolutionary Design	Build displays from existing parts

cockpit objects. The 'vast storehouse' of objects already exists, the designer simply assembles the objects to form the design.

In order to support an evolutionary mode of operation, the prototype ALC should provide the capability to save and retrieve cockpit displays and cockpit symbology. The symbology should consist of sets of standard cockpit instruments, specialized cockpit objects, and user defined objects. The user should also be able to add, modify, and delete cockpit symbols interactively.

The functional requirements of the ALC prototype software are summarized in Table V.

User Interface

The user interface has been touted as being "the single most important consideration in designing any computer system" [Singh, 1983:55]. It functions as a communication channel between the user and the system. The success or failure of a system often depends on the users' acceptance of the interface. It thus becomes imperative that the user interface be viewed as an aid by the user rather than a hinderence [Singh, 1983:55]. To accomplish this objective several fundamental requirements have been levied on the user interface. These requirements are described in the following paragraphs.

Ease of Use. The user interface should be easy to use. The prototype ALC is expected to be used by a variety of users with differing backgrounds. It thus becomes essential that the user interface is not tailored towards any specific group of users. The user interface should not require expertise in computer programming or computer graphics to use.

Minimal Memorization. To ensure ease of use, the user interface should minimize the commands the user must remember. Commands should be easy to understand and be displayed for user selection. Help and memory aids should also be available.

Easy to Learn. The user should not have to spend hours learning to use the system before becoming productive. A good criteria to use to measure the ease of which a user can learn to use a system is known as the '10-minute' rule [Rubinstein and Hersh,1984:8]. This rule states that it

should not take the user longer than 10 minutes to become familar and proficent with the system. If it does, the user interface should be re-evaluated.

Apply Experience. Experience acquired in one area of the system should be applicable to other contexts. The user should not have to re-learn how to use the system every time he switches applications.

Composition. All cockpit display composition should be handled by the user interface and performed on the screen. Instead of performing the design layout on paper then translating it onto a screen, composing it on the screen first will provide a direct mapping from design to implementation. The designer builds the actual display instead of a blueprint of it.

Feedback. Feedback informs the user what the system is doing. Feedback should be immediate, and where appropriate, visual.

Supportive of the design process. The user interface should interact with the user via a problem space that is familiar to the user. In the case of the prototype ALC, the problem space is a cockpit display. The user interface should allow the user to manipulate a representation of the cockpit display as if it were an actual cockpit display. The user should work directly on the task (design process) without the user interface distracting from the process [Shneiderman, 1983: 63]. The user interface should be transparent to the user. No distinction between the screen and an actual cockpit display should exist.

The user interface requirements are summarized in Table VI.

TABLE VI

User Interface Requirements.

Requirement	
Ease of Use	Does not require expert users
Minimal Memorization	Minimize command recall
Easy to Learn	10 Minute Rule
Apply Experience	Knowledge gain in one area should apply to others
Composition	All display construction performed on the screen
Feedback	Immediate and Visual
Conducive to the Design Process	ALC display viewed as an actual cockpit display

Summary

This chapter has presented the functional requirements for the ALC prototype. These requirements were divided into three areas, hardware, software, and the user interface. Hardware requirements pertain to the characteristics of the host system to support the ALC prototype. Four areas were specifically addressed, namely; display technology, input devices, output devices, and processing and storage capabilities.

Software requirements addressed the need to follow sound software engineering principles in the design of the software. Specific functional requirements of the ALC prototype were also outlined. User interface

requirements were presented separately from the functional software requirements to emphasize their importance to the ALC prototype effort. Several key requirements were outlined. The remaining chapters of this thesis apply the requirements defined in this chapter to the design and implementation of the ALC prototype.

III. System Design

The function of the ALC prototype is to provide an environment that supports the creation and modification of cockpit layout designs. The requirements for this environment were presented in Chapter 2. This chapter provides a system level design of the ALC based on those requirements. The design approach used is first described, followed by a system design based on that approach.

Design Approach

This thesis employed the concepts associated with object-oriented design (OOD) as the primary philosophy driving the design of the prototype ALC software. This particular design methology was chosen because it provides a direct means of mapping the problem space onto a representation of the program space (i.e. the implementation) [Booch, 1983:40]. The designer can define and manipulate representations of entities (i.e. objects) in the program space as though they were in the problem space. A one-to-one mapping is maintained between the problem space and the implementation. The reader should refer to Appendix A for a further discussion of OOD and its relation to more traditional design methodologies.

Object-oriented design starts by restating the problem in terms of its requirements. This is then followed by a conceptual implemention in which object and associated operations are identified. Objects being the actual entities that populate the problem space and operations being the actions that manipulate the objects. Once identified, the objects and corresponding

operations are then implemented (i.e. mapped to a software realization). These implementation details are not addressed in the design process, but rather discussed in Chapter 7.

System Design

As stated previously, the ALC prototype is intended to be a highly interactive graphics system capable of supporting the creation and modification of cockpit layout designs. To accomplish this goal, several operational requirements were levied on the prototype ALC. These requirements addressed the need of the prototype ALC to support:

- a user interface that is supportive of the design process,
- the dynamic creation and modification of cockpit layouts,
- the dynamic creation and modification of cockpit symbology, and
- an archival mechanism for storage and retrieval of cockpit layouts and symbology for future use.

These requirements can be realized as four separate but interrelated functional components: 1) the user interface, 2) Layout Editor, 3) Symbol Editor, and 4) a Librarian. The specific functions that each of these components perform will be addressed in later chapters (4, 5, and 6). This chapter serves to identify these four major functional components and briefly to describe their interaction.

The interaction among these components is best viewed by using a layered approach. Figure 3. illustrates the interrelationship of these components based on this approach. Each layer represents the relative degree of interaction among the components. Only those components that

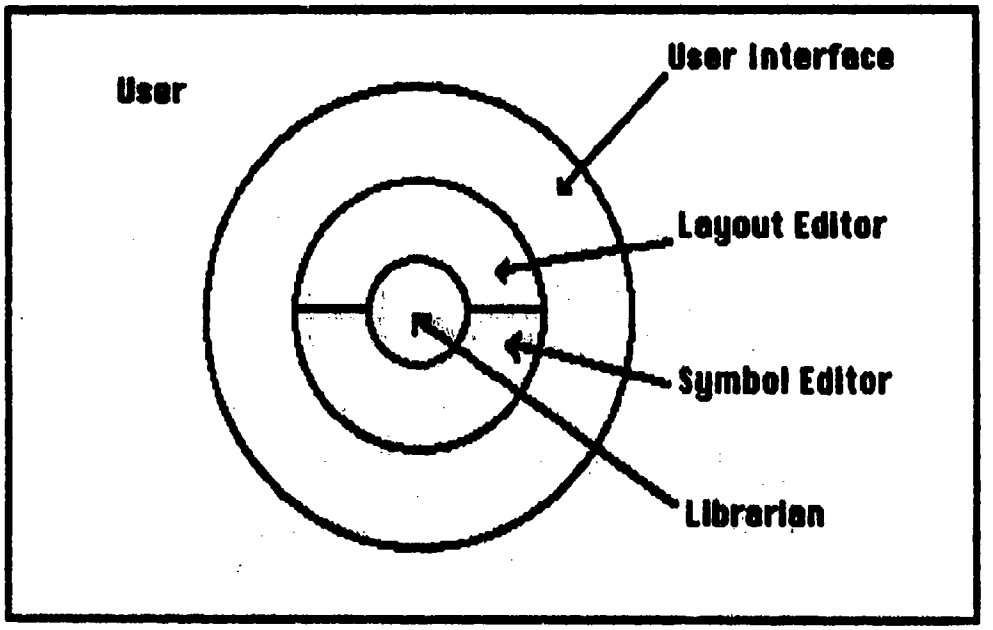

Figure 3. Prototype ALC System Structure.

share a common boundary can interact directly with each other. A description of each layer follows.

The most visible layer of the structure is the user interface. This is the only layer with which the user directly interacts. All user requests and/or system responses are conveyed through this layer. The user interface maintains a consistent and familiar boundary between the user and the other layers.

The next layer in the structure contains the Layout and Symbol editors. The Layout Editor supports the creation and modification of cockpit layout designs while the Symbol Editor supports the creation and modification of individual cockpit symbols. Both editors are directly accessible from the user interface and both can directly access the Librarian.

The last layer, the Librarian, serves as a repository for all cockpit layout designs and cockpit symbology. The Librarian can only be accessed by

one of the two editors. The user cannot directly access the Librarian without invoking an editor first. This prevents the user from unwittingly corrupting the cockpit layouts or symbology.

The detailed design considerations for each layer will be presented in the following chapters.

Summary

The design of the prototype ALC software is based on an object-oriented design (OOD) approach. Basically OOD involves,

- defining the problem,
- identifying the objects in the problem domain, and
- identifying the operations on those objects.

This approach was then used to derive a system level design of the ALC prototype. Four system components (objects) were identified. They were the user interface, Layout Editor, Symbol Editor, and the Librarian. The interrelationship of these components was illustrated. The following chapters (4, 5, and 6) apply this design method to each of the system components identified, providing detailed design considerations for the prototype ALC software.

IV. User Interface

The most visible aspect of the ALC prototype is the user interface (Figure 4). It is the medium through which all user actions and system responses are handled. The style in which the interaction is conducted can directly influence the user perception and acceptance of the system. It thus becomes imperative that the user interface is based on ideas or concepts that enhance the design process. The user interface should be viewed as an aid rather than a hindrance. The discussion of the design of the prototype ALC user interface is divided into two sections. The first section describes key design considerations that were used during the interface design phase. The second section presents the realization of these considerations embodied in the actual user interface components.

Design Criteria

Traditionally, computer systems were designed to give the user the utmost amount of computer power. Little, if any, attention was given to user interface issues. As a result, users were often overwhelmed with complex and terse commands. Initial user productivity suffered due to the level of training required to become proficient in system use. Procedures and concepts learned in one section of the system seldom carried over to others [Harslem,1984:105]. Users were often confronted with systems that were cumbersome, frustrating, and difficult to use [Bertino, 1983:38].

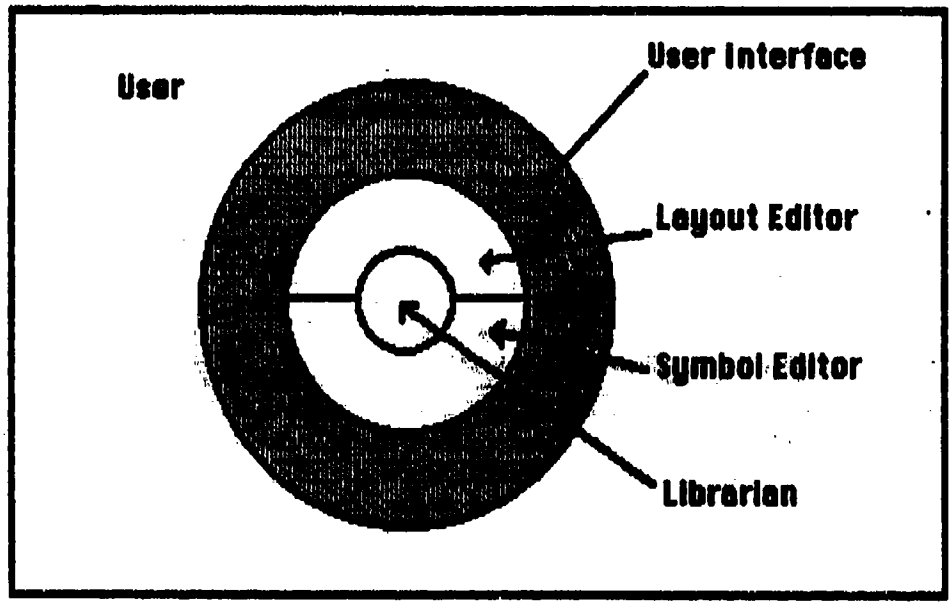

Figure 4. User Interface Layer.

Recent trends in system development, most noticeably the Xerox Star [Smith et. al., 1982] and Apple Lisa and Macintosh [Apple Computer Inc., 1985], have shifted the attention given to the user interface issues to the forefront of the design process. The user interface is no longer considered an 'add-on' component, but viewed as the single element that binds the system and the user into a cohesive whole.

To ensure a useful interface, several design considerations were followed. These design considerations were:

- Familiar User's Conceptual Model
- Supportive Dialogue Mode
- Visual Fidelity
- Consistency

Familiar User's Conceptual Model. A user's conceptual model defines the set of concepts that explain the behavior of the system [Smith et. al., 1982:248]. Specifically the conceptual model:

(1) defines the general form of the set of capabilities perceived by the user.

(2) gives the philosophy behind the system, ideally in a manner that the user is both familiar with and comfortable with.

(3) develops, in the user's mind, a framework of the system which the user should be able to associate with and which he/she can learn, comprehend and use to interpret the system's behavior. [Bertino,1985: 38-39]

Newman and Sproull have likened the conceptual model to that of a grammar for a foreign language. Their premise being, like a foreign grammar that defines the rules of communication, the conceptual model defines the way the user will perceive the interaction with the system. Fluent communication is achieved only when the model or the grammar becomes second nature. The model is not perceived as a guiding influence but rather as being 'installed' in the mind of the user [Newman and Sproull, 1979: 448].

The development of the conceptual model can significantly influence the design of the user interface. There are two basic approaches to the development of conceptual models, innovation and emulation [Bertino,1985: 39]. Innovation models exploit new types of representational possibilities in the user interface. They introduce new ways of thinking about situations and new procedures for dealing with them. Emulation models, on the other hand, emulate the current activities employed by the user in an existing system. Familiar concepts, knowledge, and procedures are incorporated into the user interface. This usually makes the model more intuitive and easier to learn.

The ALC prototype takes an emulation approach to the development of its conceptual model. Emulating the current design process, the conceptual model incorporates as its central theme the idea of constructing cockpit layouts in a building block fashion. The building blocks are pre-defined sets of cockpit components that actually represent those being used in the design process. The procedures used and the cockpit components remain familiar to the user. The user is not required to develop a new mindset to interact effectively with the user interface.

A convenient way of representing such a model is via the use of an object-oriented paradigm. The model is viewed as a set of objects (cockpit components) and a set of operations (selection and placement) that manipulate the objects and its environment (cockpit layout) [Newman and Sproull, 1979:448; Hearn and Baker, 1986:330]. Models based on such a paradigm provide an effective means of representing the problem space. As Cox points out:

> Objects are natural metaphors for model building in that each is a capsule of state and behavior, a virtual machine that can be used as a computer-based executable instance of a corresponding entity in the user's problem domain. This potential for close correspondence between computer and problem domain can be useful in building inexpensive, understandable systems [Cox, 1984:57].

Such models allow the user to interact directly with the objects of interest without concern for the actual object implementation [Arora et. al., 1985:465]. The user is not forced to interact with the system in computer terms (thus avoiding detailed procedural specifications) [Foley and McMath, 1986: 16] but rather interacts at the display level, where ideas and concepts

can be formalized and tried [Glinert and Tanimoto, 1984:11]. The objects and associated permissible actions become the interface [Arora et. al., 1985:465]. The distinction between design and implementation fades.

The perceived ability to interact directly with the objects lends itself naturally to the cockpit design process. Objects, in this case, manifest themselves as cockpit entities. Operations are provided that allow direct spatial manipulation of individual cockpit entities and direct spatial and structural manipulation of individual cockpit entity attributes. The actual implementation of each cockpit entity is hidden from the designer, only the entities behavior is observable.

Accepting an object-oriented viewpoint allows the designer to visualize the interface as a virtual cockpit display. Virtual in the sense that a multitude of cockpit display types, consisting of many cockpit entities, can be created via the same interface and methodology. The methodology simply being the selection and placement of cockpit entities on the cockpit display. Such a model provides the designer a familiar and workable environment for cockpit display generation.

Supportive Dialogue Mode. There are two basic dialogue modes: user-initiated and system-initiated [Singh et. al.,1983:56]. The choice of which dialogue mode to use depends on the intended audience and the type of interaction desired.

User-initiated dialogues require the user to issue commands in order to accomplish a task. The user is responsible for memorizing command syntax and issuing commands at the appropriate time and in proper sequence. Little if any prompting is performed by the interface. This mode of dialogue provides the greatest degree of flexibility but places an extra burden

(command memorization) on the user. It is best suited for expert users of a system [Hearn and Baker, 1986:333].

System-initiated dialogues require almost no memorization. They display all relevant information pertaining to the task at hand and prompt the user for command selection. They are best suited for novice users [Hearn and Baker, 1986:333] since they guide the user through the command selection and sequencing. This type of dialogue relies on recognition of commands rather than recall.

A system-initiated dialogue was chosen as the dialogue method for the prototype ALC user interface. System-initiated dialogues are better suited than user-initiated dialogues for the user interface requirements identified in Chapter 2. Specifically, a system-initiated dialogue approach satisfies the requirements for ease of use, minimal memorization, and ease of learning.

System-initiated dialogues tend to be more novice-oriented than user-initiated dialogues [Hearn and Baker, 1986:333]. They assume little, if any, prior technical expertise on the part of the user, lending themselves naturally for use by a large audience. A prime requirement of the ALC is that it be usable by a variety of users, with different technical backgrounds. It is essential that the user interface is not tailored toward any specific group.

System-initiated dialogues minimize the amount of memorization required by the user. All objects and commands of interest are displayed and available for selection. The user is not required to remember command syntax or command sequencing. Relieving the user of this burden directly impacts the quality of the thinking (i.e. design) process.

Studies have shown that "conscious thought deals with concepts in short-term memory and the capacity of short-term memory is limited" [Smith et. al.,1982:260]. By displaying available commands, short-term memory is relieved of the burden of command recall and syntax formulation. Thinking becomes easier and more productive as the user is permitted to concentrate on the creative aspects of the design process without being burdened by the dialogue [Smith et. al., 1982:260]. The dialogue becomes a mechanical device for issuing actions without impacting the conscious thought (design) process [Bertino,1985: 50].

Learning is also eased by system-initiated dialogues. Learning in this context, refers to the time it takes a user to become proficient in a system's use, in order to perform productive work. This is not to imply that the users needs to be proficient in all aspects of system use; they seldom are [Rubinstein and Hersh, 1984:8]. But rather, the user needs to know only the subset of the system that directly influences the task at hand. The time it takes to learn these capabilities should be minimial. A general rule of thumb, known as the '10 minute' rule, attempts to limit this learning time to ten minutes. If it takes longer than ten minutes, the interface should probably be re-evaluated and perhaps redesigned.

System-initiated dialogues provide a viable means of satisfying the ten minute constraint. Information needed to converse with the interface is always displayed, relieving the user of the burden of command syntax recall. The user can experiment with commands and options immediately without worrying about command syntax or key-in errors. Attention is focused on system understanding instead of being divided among tasks. A semantic understanding of the system (concepts and functionality) is gained rather than a syntactic (detail) [Shneiderman, 1983:65].

System-initiated dialogues also aid in regaining proficiency in a system after an extended absence. It is desirable to have the user 'up to speed' as soon as possible. Since the user has already developed a semantic understanding of the system and its commands, the time required to become proficient will usually be less than if the system were based on a user-initiated dialogue. The reason being, syntactic knowledge is "volatile in memory and is easily forgotten if not frequently used" [Shneiderman, 1983:65]. Semantic knowledge tends to be more system independent and once "acquired through general explanation, analogy, and example, is easily anchored in familiar concepts and is therefore stable in memory" [Shneiderman, 1983:65]. The stability of semantic knowledge allows a user to regain proficiency faster and retain it longer.

Visual Fidelity. Visual fidelity or "What You See is What You Get" refers to the ability of representing a rendition of the actual output on the display screen [Smith et. al., 1982:264]. In this application area the display screen is the computer display that the user sees, and the output is an actual cockpit display that the pilot sees.

Visual fidelity provides the user with a means of directly interacting with the problem space; seemingly by-passing the user interface [Shneidermann, 1983:63]. "The user operates directly on the data in a form convenient to him *(i.e. cockpit objects)*, not one imposed by the computer *(i.e. code)*" [Harslem, 1984:103] (italic phrases added by author). Cockpit layouts are composed directly on the display screen and mapped directly to the target cockpit display. The user's view of the display screen and the cockpit display are inseparable.

Consistency. Consistency in the user interface allows the user to interact with various parts of the system without having to change the method of interaction. The user learns one method of interaction as opposed to several. A consistent interface reduces the amount of re-learning that user must perform while switching between applications [Harslem, 1984:105]. It allows the skills, procedures, and concepts acquired in one section of the system to be applied equally to other sections [Marcus, 1984:24].

The prototype ALC promotes consistency by providing a single user interface for all sections of the system. Commands are generic in nature, thus allowing for a small set of commands to be used throughout. The extraneous application-specific semantics of a command are stripped away allowing the user to deal directly with its underlying meaning [Smith et. al., 1982: 268]. The actual physical interaction (object positioning and selection, command selection) is performed via a mouse. The keyboard is used exclusively for textual input only.

Table VII summarizes the design considerations which guided the design of the prototype ALC user interface.

User Interface Components

Based on the design considerations discussed above and the user interface requirements presented in Chapter 2, the user interface design of the prototype ALC can be satisfied by providing five components, the keyboard, a mouse, windows, command bottons, and menus. A description of each component and its associated functions follow.

Table VII

User Interface Design Criteria

Criteria	
Conceptual Model	Emulation-Mode, models the current design process
Dialogue Mode	System-initiated
Visual Fidelity	'What You See Is What You Get'
Consistency	Single User Interface Generic Commands

Keyboard. The keyboard is an input-only device used exclusively for textual input. Special characters, control character sequences, or command sequences are ignored by the user interface. The user interface only recognizes input from the keyboard when a request is made for textual data.

Mouse. A mouse is used as the primary physical means of interacting with the interface. The mouse serves as an extension (prosthesis) of the user, allowing the user to point to a location on the screen and make a selection.

Most mice available today support a number of buttons for object selection. Often an application will assign different functions to these buttons. The prototype ALC user interface treats all buttons on a mouse as a single pick device. No matter which button is pressed, the result is a

simple pick/selection action. Restricting the function of the buttons aids in the portability of the interface and prevents the user from making a mistake (i.e. pressing the wrong button).

Window. Windows are rectangular regions on the display screen which serve as the focal point for user interaction. The user interface supports three types of windows, applications, option, and dialogue.

Application windows provide a medium for viewing and manipulating objects. All objects (cockpit displays and symbols) that the user can manipulate are displayed within application windows.

Option windows display representations of objects that are available for selection. An active option is indicated by highlighting the option selected.

Dialogue windows serve a multi-purpose role. First, they provide a standard means of requesting additional information pertinent to the execution of a command (i.e. prompting for file name on a 'SAVE'). Second, they serve as a means of confirming the intention of the user (i.e. queries to insure the user really wants to delete a file). Lastly, the dialogue window is used to inform the user of any error conditions.

Application and option windows are classified as 'modeless' windows, meaning that their presence does not require a response by the user. The user is free to interact with these windows as he chooses. On the other hand, the dialogue window is known as a 'modal' window. This window, when displayed requires the next user response to be directed toward it. The user is forced into a response mode. The user must satisfy the window prompt before futher processing can take place. The mouse and the keyboard are dedicated to the dialogue window.

Command Buttons. Command buttons act as screen function keys. They serve as a means of issuing commands to the system. Only those commands that are pertinent to the current processing are displayed.

Menu. Menus serve as another medium for command invocation. Menus consist of logically related commands, herein refered to as menu items.

The user interface supports what are known as 'pop-up' menus. The term pop-up comes from the way these menus are activated. Initially, the user only sees a menu title on the screen. If a title is selected, the menu items associated with that title 'pop-up' underneath it. The cursor can then be moved over the desired item for selection. When a selection is made or the cursor is moved outside the menu region, the menu disappears.

Pop-up menus force the user into a response mode only when they are activated. At which time, the interface directs all user actions toward menu selection. The user is required to make a selection or cancel the menu activation by moving the cursor outside the menu region. All other user actions are ignored when a menu is activated. Otherwise, pop-up menus impose no restrictions on user actions.

Summary

The user interface has often been considered the "most difficult and the least understood part of interactive systems" [Singh et. al.,1983:55]. This chapter has presented the user interface in two parts. The first part described the design considerations that were adhered to in conceptual design of the interface. The second part described the actual components that constitute the physical structure of the interface.

V. Editors

The ALC prototype incorporates two distinct, yet similar, graphic editors to enhance the design process (Figure 5). The Layout Editor, directly supports the creation and modification of cockpit displays. The second editor, the Symbol Editor, supports the creation and modification of individual cockpit symbols. The editors are similar in the methodologies they employ, but differ on the view of the design process they present to the designer. The Layout Editor allows the designer to view and manipulate a design as a collection of spatially arranged cockpit symbols, while the Symbol editor allows the designer to view and manipulate the internals of individual cockpit symbols. The similarities and differences of these two editors are presented in this chapter.

Editor Similarities

Currently, cockpit display construction starts with the design and analysis of the cockpit layout on paper. When complete, the design is transferred to software experts for translation into code. Depending on the complexity of the design and the amount of new code needed to be generated, the designer may not realize the implementation of the design for weeks or months after submission. To make matters worse, modifications to the implemented design could entail further delays. This process is very time consuming and programmer-intensive. The designer's productivity is at the mercy of the software development process.

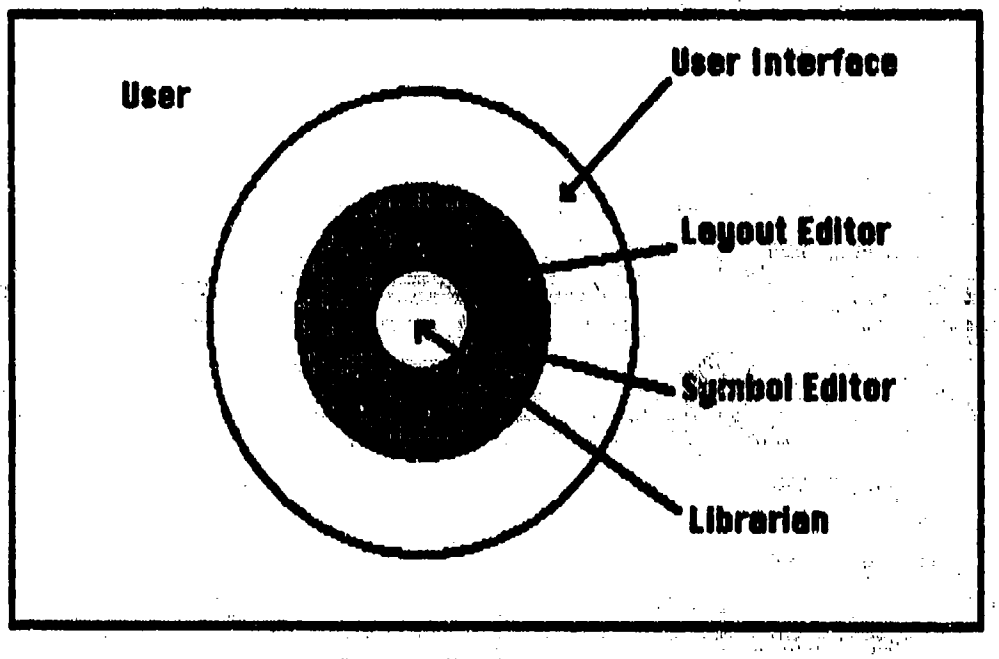

Figure 5. Editor Layer.

To combat this problem, the ALC prototype incorporates two types of editors, the Layout Editor and the Symbol Editor. Both editors are graphics based and object-oriented in nature. Their function is to make the design and analysis phase a more productive and enjoyable task by supporting the bulk of the design activity in a responsive, interactive graphics environment. The goals of these editors are to free the designer from the tedium of designing cockpit displays on paper and reduce the need for software experts for design implementation. To accomplish this, the editors promote the interactive design of cockpit displays and symbology from pre-defined classes of cockpit symbols. The Layout Editor supports the design of cockpit displays, while the Symbol Editor supports cockpit symbology construction.

Although both editors perform different tasks, there are four common requirements that both must support:

- Graphic Interaction,
- Iterative Development,
- Evolutionary Design,
- Experimentation.

Graphic Interaction. The process of designing a cockpit display is a spatially oriented activity. It relies heavily on the use of graphic images for representing cockpit and real-world objects. Both editors exploit the use of graphics as the main medium of interaction. All editing is performed on graphic entities versus the use of textual information. The editors work directly in the mode most natural to the design process, graphics.

Iterative Development. Iterative development is a process by which designs are continuously evaluated and modified to satisfy design requirements. Both editors take on an active role in supporting this requirement. Both editors assist the user by providing the facility for creating new designs, modifying existing designs, and supporting the rapid generation of alternative designs.

The primary function of both editors is the creation of new cockpit designs. Designs are created by selecting and arranging (inserting) pre-defined objects on the display screen to satisfy a design requirement. Besides inserting objects, both editors also support object deletion and repositioning. Final designs, or work in progress, can be saved for future use or discarded.

In addition to creating new designs, the editors also support the capability of modifying existing designs or generating alternative representations of existing designs. Modifying a design usually involves the inserting or deleting objects from a layout, then saving the result. This in affect destroys the original design. Creating an alternative design representation is a non-destructive editing procedure. The original design serves as a template from which objects can be added or deleted. The modified original is saved as a different design, leaving the original design intact.

Evolutionary Design. The premise behind an evolutionary design approach is to eliminate the practice of 'reinventing the wheel' every time a design is created. To accomplish this, the ALC prototype promotes the creation of designs from pre-existing sets of cockpit symbols. Designs can then be constructed in a building block fashion from these symbols. The editors directly support this methodology by allowing the user to manipulate these symbols as discrete entities; they can be selected, positioned, and deleted from the display. The user never deals with anything conceptually simpler than an object.

Experimentation. Both editors actively support experimentation. They provide the designer with the capability of experimenting with different design layouts without committing the work to a final design. To support experimentation, both editors allow the user to undo the last operation performed, erase the entire design, and restore the design to its original form.

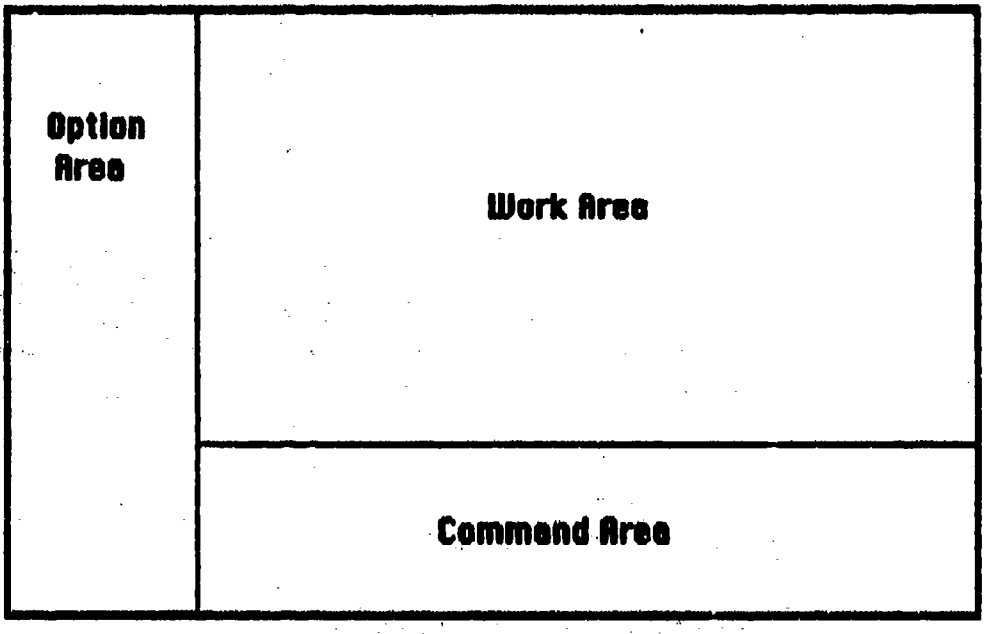

Figure 6. Generic Editor Layout.

Functional Similarities

The similarities of the two editors are best delineated by identifying the functional areas that comprise each editor. This is conveniently done by illustrating the generic display form common to both editors (Figure 6). The display consists of three functional areas; the option area, the work area, and the command area.

The option area of the display, maintains a list the objects that are currently available for selection. The objects are pictorially displayed in miniature form. The process of object selection is identical for both editors.

Objects are selected by positioning the cursor over the desired object and pressing a mouse button. The selected object is 'made active' and is highlighted to indicate selection. The next time the cursor is within the work area and a mouse button pressed, the selected object will be drawn. The selected object remains active until another object is selected or the work

area is cleared. This allows the user to add multiple images of the selected object on to the work area without having to reselect the object each time.

The second functional area of the display is the work area. This area serves as the designer's chalkboard. Designs are constructed and displayed within this region. Within this area the user can perform three basic functions; object insertion, object deletion, and object repositioning. The process of insertion is unique to the editor used and will be cover in detail in later sections. Deletion and repositioning, however, are identical to both editors and will be addressed here.

Deletion is the process of removing an object from the work area. Objects are removed in a manner analogous to option selection. First the object is selected, then the 'Trash' button is selected. This cause the selected object to be removed from the work area and the work area to be redrawn.

Repositioning is also similar to option selection. First the object is selected within the work area, then the cursor is repositioned and a mouse selection is made. The selected object is erased from its original location and redrawn at the new. If the new location is outside the work area the object remains in its original location.

To aid in the deletion and repositioning process, an extent box is drawn around the selected object. The extent box serves to identify exactly which object is selected. This is helpful when multiple objects overlay each other, or are positioned in close proximity to each other. The user is not left guessing which object was actually selected, but can directly determine by visual inspection. Besides identifying an object, the extent box also tracks the cursor while inside the work area. This provides the user with a spatial reference for repositioning.

The third area of the editor displays is the command section. All commands that effect the editor are listed here. Commands are displayed as command buttons. The user selects a command by positioning the cursor over the desired button and pressing a mouse button. The editors interpret the selection and perform the desired action.

There are several commands that are identical to both editors. These commands are listed below with an accompanying description. Commands unique to an editor will be presented under that editor's description.

New: Resets the editor by clearing the current design. If the current design is not saved prior to issuing this command, the editor prompts to save it.

Old: Retrieves an existing design. If the current design is not saved prior to issuing this command, the editor prompts to save it.

Save: Saves the current design. If this is a new design the user is prompted for a title, otherwise the design is saved under its original title.

SaveAs: Saves the current display under a different title.

Undo: Undoes the last operation performed in the work area.

Clear: Clears the work area and de-activates the currently selected object.

Revert: Restores the current design to its original content. If the current design is new then this command has no effect.

Quit: Terminates the editing session and exits the user from the environment.

Although the editors are similar in many respects, there are important differences that distiguish the two editors. These differences are presented in the following sections.

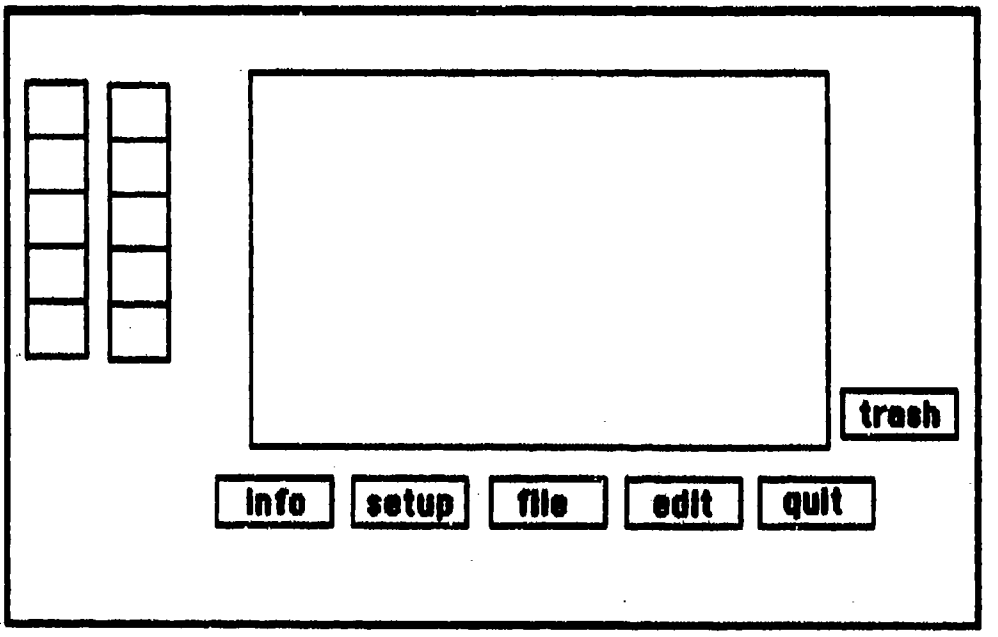

Figure 7. Typical Layout Editor Format.

Layout Editor

The Layout Editor serves as the sole means of creating and modifying cockpit layout designs. Layouts are constructed in a building block fashion from sets of predefined cockpit components or symbols. The user simply selects the desired component, from an available option list, and places it on the cockpit layout. A typical format of the Layout Editor is provided in Figure 7. The functional capabilities unique to the Layout Editor are discussed in the following paragraphs.

The option area of the Layout Editor differs from the Symbol Editor in one major respect; namely, the types of objects displayed for selection. Because the Layout Editor is intended to be used to create a multitude of different cockpit layouts, the objects available for selection will vary with the design being constructed.

The Layout Editor handles object insertion differently than the Symbol Editor. Basically, object insertion is based on a technique known as 'dragging'. Dragging is an interactive technique for dynamically moving an object under cursor control. Objects selected in the option area are 'dragged' into the work area, positioned, then inserted by pressing a mouse button. To aid the user in object placement, an extent box, defining the object's size, is drawn around the cursor. The extent box tracks the cursor as it is moved around inside the work area. When the object is inserted, the extent box is removed.

The Layout Editor supports two unique commands; namely,

Info: Displays a narrative of the selected object. This command is ignored by the editor if there is no object selected.

Set Up: Provides a means of accessing the cockpit symbology libraries. The user is provided with the options to select new classes of symbols, remove current classes from the display, or invoke the Symbol Editor to modify a cockpit symbol.

Symbol Editor

The Symbol Editor provides the user with a means of creating and modifying cockpit symbology. Symbols are created in a manner analogous to cockpit layouts, except the range of options to choose from is limited and each object has its own unique method for being inserted into the work area. Figure 8 illustrates the display format for the Symbol Editor.

Unlike the Layout Editor, whose options are interchangable sets of cockpit components, the Symbol Editor maintains a single set of options. This set consists of six graphic primitives: point, line, rectangle, circle, polygon, and text. All cockpit symbols are constructed from this set.

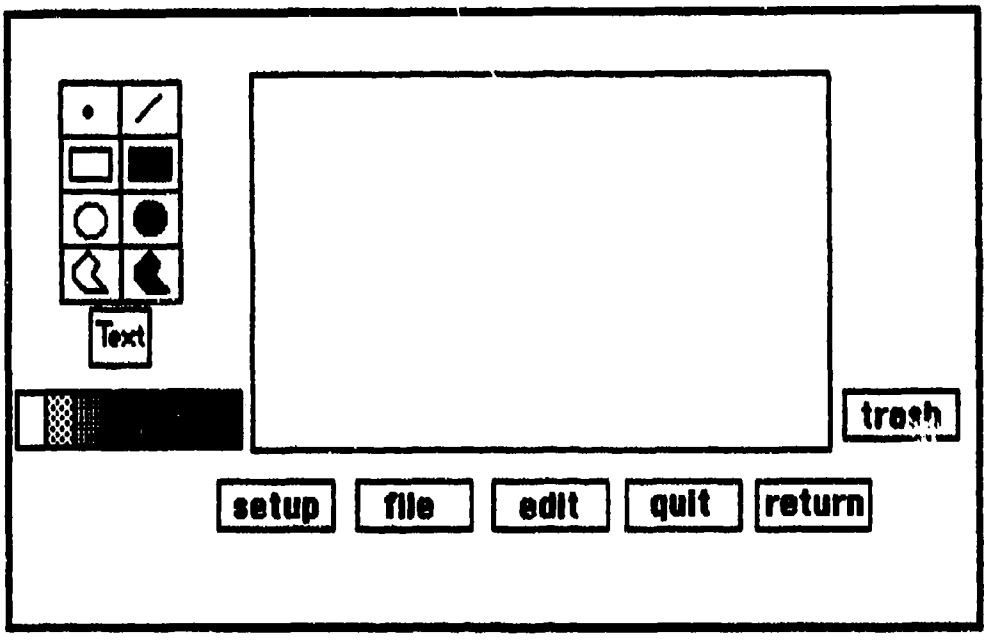

Figure 8. Typical Symbol Editor Layout.

The method of inserting (placing) objects in the work area also differs between the two editors. In the Layout Editor, object placement is determined with a single mouse selection. This works for all objects. Object placement in the Symbol Editor is a bit more complicated. Each graphic primitive (object), because of its unique geometric form, requires a different placement method. For example, points are positioned in the work area with a single mouse selection, where as, lines, rectangles, circles, and polygons, require multiple insertion points to be defined. The methods for inserting these primitives will be presented next.

Defining multiple points for primitives is accomplished using a 'rubberband' technique. Basically, rubberbanding involves defining a starting point for an object, followed by moving the cursor to define other points. As the cursor is moved, the object is streched between the initial point and the cursor's current position. This dynamically alters the shape of the object providing the user with immediate feedback about the object's shape.

Using this technique, procedures for adding lines, rectangles, circles, and polygons can be defined.

Lines, rectangles, and circles are added to the work area by defining two points that bind the primitive to the work area. In the case of the line primitive, the first point defines the starting point while the second point defines the end of the line. The rectangle primitive uses the first point to defines its lower left hand corner. The upper right hand corner is then defined by the second point. The circle primitive uses the first point to define its origin, and the second point to define its radius.

Polygons differ from the other primitives, in that they require multiple points to define their shapes. Polygons are drawn by specifying a series of connected line. Each line forms an edge to the polygon. The user needs only to specify the starting endpoint once, afterwards the endpoint from the previous line is used as the new starting endpoint. Polygon drawing is terminated when the user selects a point outside the work area or another option is selected.

Text is the only primitive that requires the use of two input devices, mouse and keyboard. The mouse is used to select the start point in the work area where the text is to be inserted. The keyboard is then used to enter the actual text. The insertion point must be selected before the text is entered, otherwise the interface will ignore any text input.

The Symbol Editor has one command unique to its processing which is:

Return : Transfers control back to the Layout Editor.

Summary

Two graphic editors, the Layout Editor and the Symbol Editor, are used to support the design process. The Layout Editor supports the creation and modification of cockpit layout displays, where as the Symbol Editor supports the creation and modification of cockpit symbology.

Both editors have in common several functional similarities; such as, graphical interaction, iterative development, evolutionary design, and experimentation. The editors differ at the level of design abstraction at which they are employed. The Layout Editor is used at the highest level of design abstraction, i.e. cockpit layout editing. The Symbol Editor at the lowest level, i.e. cockpit symbology editing. Together they provide the design tools needed to construct cockpit layout designs in a dynamic, interactive mode.

VI. Librarian Design

To make the prototype ALC a truly functional and supportive environment for the design of cockpit layouts, a facility for storing and retrieving cockpit layouts and cockpit symbology is needed. Such a facility would eliminate the practice of 're-inventing the wheel' every time a new design is requested. The designer would have available, on-line, a means of accessing existing cockpit layouts and symbology from which the new design could take root. The prototype ALC supports such a facility, namely, the Librarian (Figure 9). The Librarian is an on-line, archival mechanism. Both cockpit layouts and cockpit symbology are supported. This chapter describes the design of the Librarian in terms of how layouts and symbols are stored, how they are retrieved, and problems relating to consistency.

Object Storage

All objects for the ALC prototype are stored in libraries. The Librarian maintains two separate libraries; namely, the Layout Library and the Symbol Library. As the names imply, the Layout Library is used for the archival of cockpit layout designs, and the Symbol Library archives all the cockpit symbols used in the system.

Libraries are subdivided into classes. Classes are logical groupings of objects. Objects are assigned classes based on their type. Partitioning libraries into classes reduces the overall complexity of the library, allowing for efficient and rapid access of individual objects.

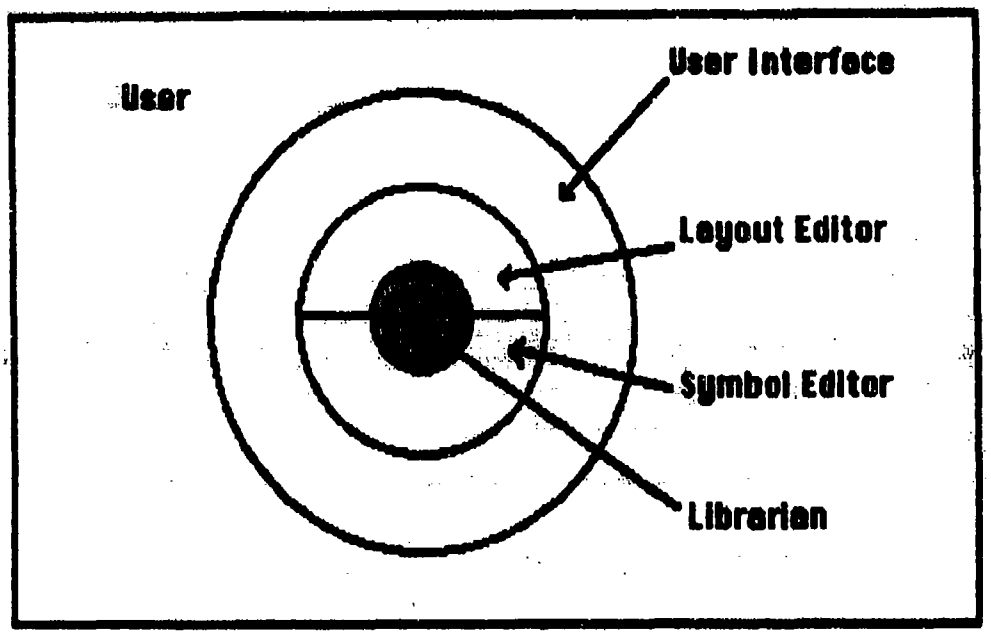

Figure 9. Librarian Layer.

The Layout Library is partitioned into classes based on cockpit display types. Figure 10 illustrates a typical Layout Library partitioned into three classes, Aircraft, HUD, and Threat. Each class in a Layout Library contains the actual cockpit layout displays associated with a particular class type. For example, the class HUD contains three HUD layouts.

The Symbol Library is very similar in structure to the Layout Library. It too is partitioned into classes, but the partitioning is based on symbol type, not layout type. Each class in the Symbol Library is further partitioned into relations. Relations are groupings of similar, yet different cockpit components. They are similar in that they are classified under the same class, but differ in content. A relation's content is composed of individual symbols defined in the class's symbol families. Symbol families are the fundamental symbol groupings in the library. Each family contains permutations of a single symbol type. For example, the family Flight Path

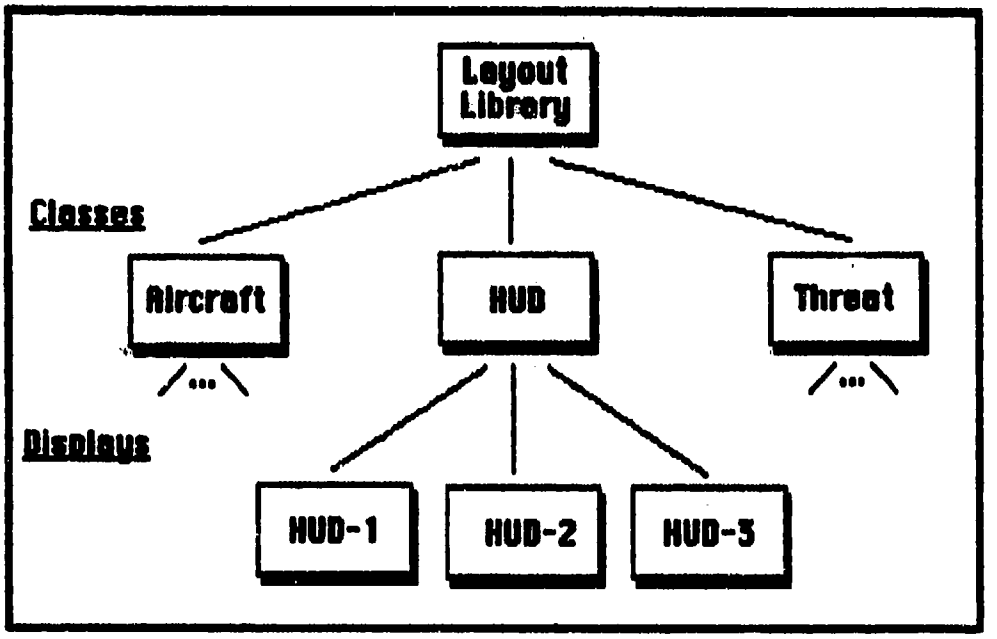

Figure 10. Typical Layout Library.

Marker contains different representations of a flight path marker. The representations differ, but they symbolize the same thing. Figure 11 illustrates a typical symbol library.

The symbol library shown in Figure 11 is partitioned into three classes; Aircraft, HUD, and Threat. These classes should not be confused with the classes defined in Figure 10. Although they have the same names, they are not the same. The classes in Figure 10 represent logical groupings of cockpit layouts, where as, the classes in Figure 11 represent the logical groupings of cockpit symbols found on different types of layout displays. For example, the symbol class HUD contains symbols relating to the construction of HUD type displays. It would not contain symbols such as aircraft silhouettes or armaments. These symbols would most likely be contained within the Aircraft class.

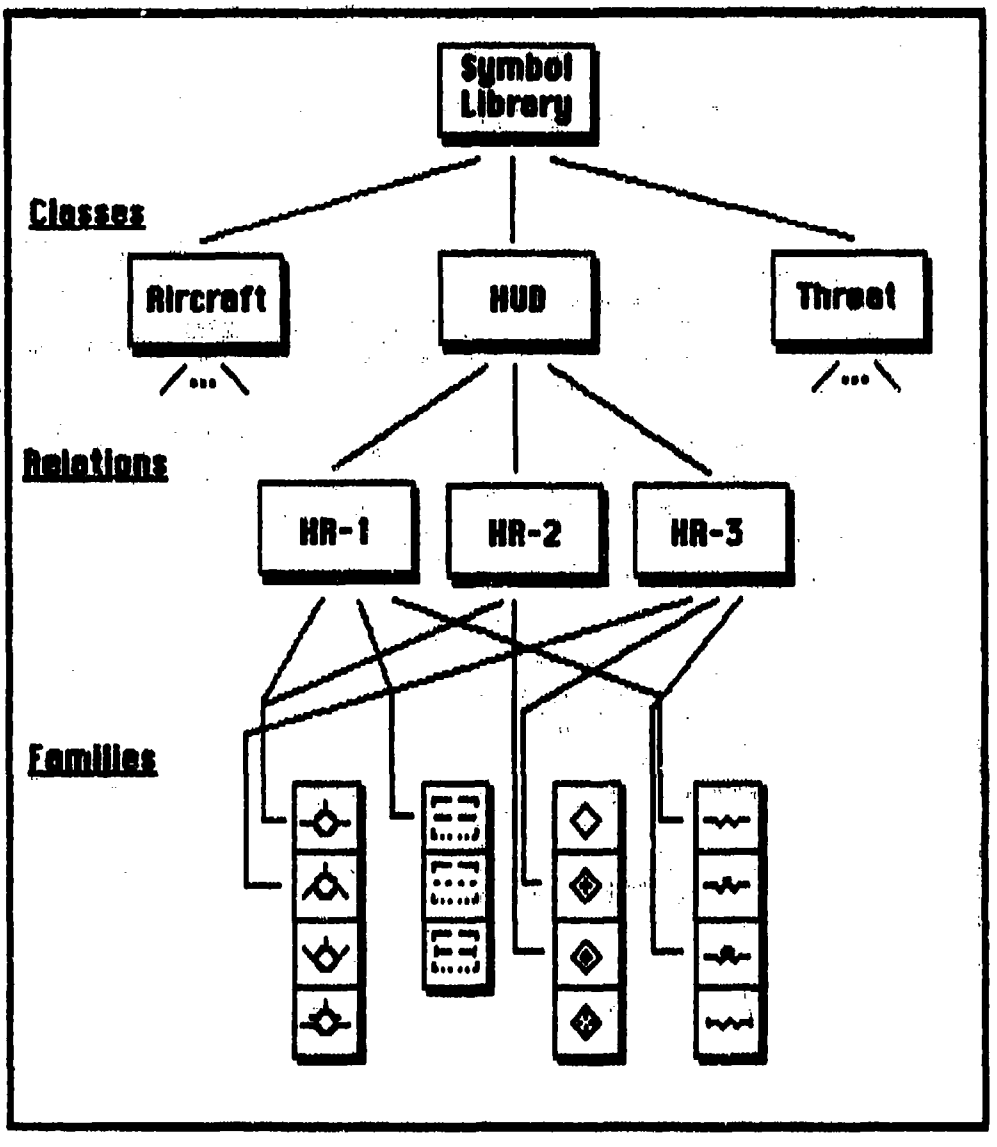

Figure 11. Typical Symbol Library.

Each class in the Symbol Library, is in turn, partitioned into relations. The class HUD contains three relations, HR-1, HR-2, and HR-3. Each relation is composed of symbols defined in the HUD class symbol families. These families, listed left to right in Figure 11 are, Flight Path Marker, Pitch Ladder, Missile Aim, and Aircraft Reference. These families of symbols define all the HUD symbols currently available for use in HUD layout construction.

Retrieval

Cockpit layouts and symbology are retrieved from the libraries by the Layout Editor and the Symbol Editor, repectively. The Layout Editor allows the user to access both layout designs and symbol relations. Symbol relations are mapped directly onto the Layout Editors display as option windows. The Layout Editor has the capability of modifying the layout design only. It cannot directly modify the contents of the option window (i.e. symbol relation). Additions or modifications to a symbol in the option window can only be performed via the Symbol Editor.

The Symbol Editor can directly access symbol relations or families. All modifications to relations are based on the content of the individual families. In order to add a symbol to a relation that symbol must first be defined within a family. Modifications to symbols are also performed at the family level. Deletions are possible at both levels. Deleting a symbol from a relation simply removes it from that relation. Deleting a symbol from a family removes the symbol from the system.

Consistency

Besides being able to store and retrieve cockpit layouts and symbology, the Librarian is also tasked with the responsibility of maintaining consistency between the two libraries. Inconsistencies between the two libraries has the possibility of developing when ever a cockpit symbol is modified or deleted. For example, if a symbol is deleted from a symbol family, its removal has the possibility of effecting all relations that use the symbol and all cockpit layouts that use the relation. The effect of

removing a single symbol results in a propagation of changes throughout the entire system. If the Librarian does not support active consistency checking, the integrity of the both libraries cannot be guaranteed.

Summary

The Librarian is an on-line, archival mechanism that supports the storage and retrieval of cockpit layout designs and cockpit symbology. Layouts and symbology are stored in separate libraries. These libraries are partitoned, based on layout and symbol type into logical groupings called classes. Each class within a library contains the actual cockpit object (i.e layout or symbol).

All classes are accessed via one of the two editors supported by the ALC prototype. The Layout Editor allows direct access to cockpit layouts and indirect access to symbol relations (option window). The Symbol Editor provides a means of adding, deleting, and modifying cockpit symbology. Changes to cockpit symbols has the potential of creating inconsistencies between the two libraries. It thus becomes critical that some type of active consistency checking is performed by the Librarian to ensure layout and symbol integrity.

VII. Implementation

Implementation is the process of mapping an abstract representation of a problem (i.e. design) into a concrete, functional model. Three areas of the ALC prototype design were implemented in this thesis, namely; hardware, environmental, and the application itself. Hardware implementation deals with defining the actual graphics system on which the ALC prototype is hosted. Environmental implementation deals with the programming environment in which the ALC prototype is to function. For this thesis effort an object-oriented environment was chosen. Finally, application implementation is the actual implementation of the ALC prototype itself. Each implementation phase will be discussed in detail in the following sections.

Hardware

The ALC prototype is implemented on a Raster Technologies Model One/25 graphics system. This system was chosen for its high resolution display, interactive capabilities, and full availability for this thesis effort. The description of this system is divided into five areas, display technology, input devices, output devices, storage and processing, and software drivers. The first four areas correspond to the hardware requirements guidelines provided in Chapter 2. The actual components that comprise these areas are illustrated in Figure 12. A description of the software drivers needed to make all the hardware components work is also provided.

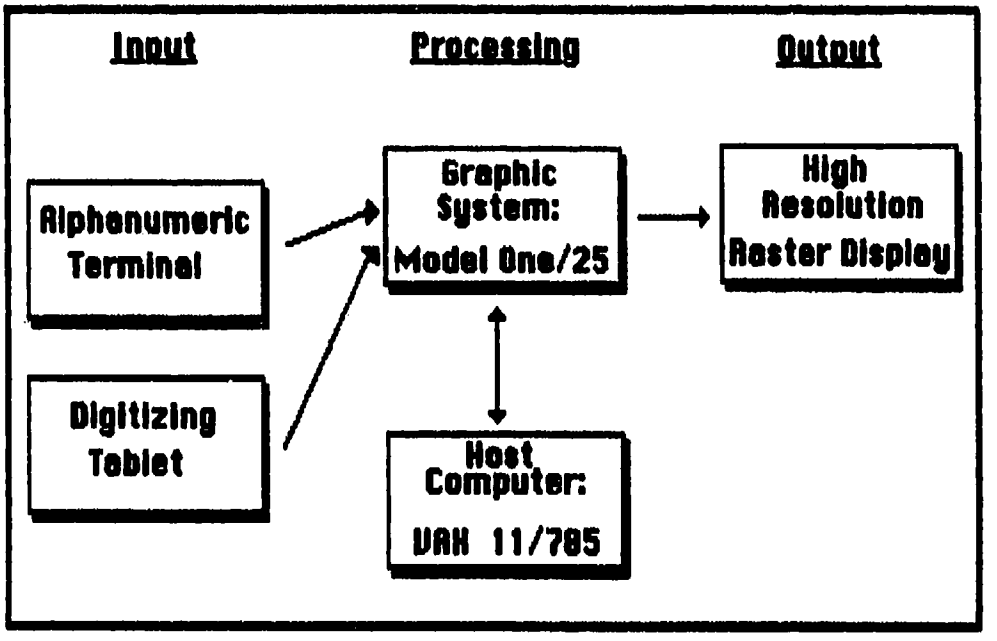

Figure 12. ALC Prototype Hardware Configuration.

Display Technology. The Raster Technologies Model One/25 employs the use of a high resolution raster display. The Model One/25 supports two levels of resolution, 512 x 512 and 1024 X 1024 pixels. In addition, the Model One/25 is capable of displaying over 16 million colors in the 512 x 512 mode. The high resolution and color capabilities of this display make it an excellent choice for hosting the ALC prototype.

Input Devices. The Model One/25 currently supports two input devices, an alphanumeric terminal and a graphics tablet with a mouse. The alphanumeric terminal is not used as an input device. It is used primarily to issue system commands to the Model One/25 and to perform system initialization (See Appendix A). Inputs from the alphanumeric terminal are ignored by the ALC prototype software.

The graphics tablet with mouse was used exclusively as an input device for the ALC prototype. All user actions (such as cursor movement and selection) are conveyed to the software through the mouse. The particular mouse used supports 16 different function buttons. All but one button (button 0) is used as simple pick devices. Button 0 is dedicated to providing the cursor with the current mouse position (this is a hardware quirk of the Model One/25). Selecting this button has no effect on processing. The other buttons when pushed, function as simple pick devices, causing a selection event to be registered in the system event queue. This allows the application software to query the event queue and process any pending events.

Output Devices. The Model One/25 currently does not support output devices other than the display screen.

Processing and Storage. The processing capabilities of the system are divided between two systems, an independent host computer and the Model One/25 display processor. All application software is developed, stored, and executed on the host computer, while all graphic operations are performed on the Model One/25. The host computer performs the actual computational tasks required of the application, passing off the graphic and interactive tasks to the Model One/25. The Model One/25 serves as an intelligent graphics terminal, interpreting and performing graphic commands sent to it by the host computer.

The host computer was a VAX 11/785, operating under BSD UNIX version 4.2. Communication between the VAX and the Model One/25 was conducted over a 9600 baud channel, routed through a local area network. The channel bandwidth seemed sufficient when the VAX was not heavily used. However, interactive response times were degraded as VAX usage increased.

Device Drivers. A problem often associated with hardware configurations as described above, is the lack of reliable device drivers to communicate with the graphics hardware. This thesis was not without such a problem. Device drivers are the actual software modules that allow a host system to interact directly with a piece of graphic hardware. They function as interpreters by translating host commands into commands understandable by the graphics hardware. The device drivers implemented in this thesis, translate graphic commands issued by the application software into ASCII character strings representing the hexidecimal value of a Model One/25 operator. This string was then sent over the network to the Model One/25 where it was interpreted and the appropriate task performed. Appendix F provides a detailed description of the device drivers that were implemented.

Object-Oriented System

This section describes the graphical object-oriented environment that was developed and implemented as part of this thesis effort to support the implementation of the ALC prototype. This environment was based on object-oriented concepts derived from such systems as Smalltalk [Goldberg and Robson,1983], Traits [Curry and Ayers,1984], Object Oriented Pre-Compiler (OOPC) [Cox,1983], and IconMaker [Kramer,1984]. The intent of this environment was not to develop a 'production quality' product, but rather a test-bed in which object-oriented concepts could be applied to the ALC prototype effort. With this in mind, this section will proceed in the following direction, first object-oriented concepts will be discussed in general terms to familiarize the reader. Next, the applicability of

object-oriented concepts to graphic systems will be presented, followed by a description of the actual environment that was implemented.

Object-Oriented Concepts. The distinction between traditional programming environments and object-oriented environments deals with the way the environment's computational model is defined. A computational model describes how the various entities in an environment interact to perform a computational task. In both environments the tasks or goals are identical; the difference lies in the definition of the entities that inhabit the environment and their method of interaction.

Traditional programming environments are based on an operator/operand model [Cox,52:1984]. This model views the computational process as being operations performed on operands. The environment is divided into two distinct sets of entities, operators (procedures) and operands (data). Operators are considered active entities in the environment that manipulate the passive data items passed to them. Operands are passive in nature, and are only changed by an operator.

Interaction between these entities is usually supported by some type of direct invocation mechanism (i.e. subroutine or procedure call). Operators are directly invoked to manipulate a set of operands. The invocation process in effect establishes 'how' something should be done. This usually places restrictions on the type of operand that an operator can manipulate. This, in turn, can populate the environment with sets of operators that conceptually perform the same operation on different data types.

An alternative to the operator/operand model is the message/operand model. This model forms the basis for object-oriented environments. Unlike the operator/operand model, where data and procedures are viewed as

separate entities, the message/operand model merges the two into a single entity referred to as an object. The object serves as the focal point for all computations. Since objects combine the properties of operands and operators, they are capable of being manipulated as well as being the manipulator [Robson,1981:76].

Objects interact via a message passing paradigm. A message is a request from one object to another to perform one of its operations. The key word in this description is 'request'. The receiver of the message determines 'how' it will handle the message, not the sender. The sender can only request 'what' should be done, it has no control over 'how'. Invocation is performed indirectly as opposed to more traditional direct invocation methods. Message passing has a direct impact on the number of operators needed to perform similar tasks. Instead of using a set of different messages (i.e. operators) to invoke a similar operation in a set of different objects (i.e. operands), message passing permits identical messages can be sent to all objects invoking a behavior unique to that object. The environment becomes more compact and consistent, since a single method is used to invoke computations instead of a set of methods.

The reader should refer to Appendix B for a detailed discussion of terminology and characteristics associated with object-oriented systems.

Graphic Systems. The use of object-oriented concepts is not new to graphic systems. Perhaps the earliest system to employ a subset of these ideas was Sketchpad [Sutherland,1980]. Sketchpad was one of the first systems to provide true interactive capabilities. Its similarity to object-oriented systems of today is found in Sketchpad's definition and creation of graphics entities (objects).

Sketchpad presented two views of objects. The first, and most intuitive, was that of a graphic entity. This entity took form on the display screen and was capable of being manipulated via interactive means. Sketchpad supported basic geometric objects such as lines, circles, and points.

The second view was actually an implementation abstraction of the first. At the implementation level, objects were quantified as being sets of variables and constraints. Variables defined the objects form, while the constraints modified the form to satisfy a given design or geometric requirement. The association of data and methods (constraints) for modifying that data is very similar to concepts found in most object-oriented systems today.

Besides a similar object definition, Sketchpad also employed an 'instantiation' technique to create objects. Objects were instances of a 'master picture'. A master picture was an original description of a specific object. Objects were instantiated by duplicating the master picture, then modifing the variables to describe the new object (instance). This is very similar to the class concept.

Sketchpad greatly influenced the interactive nature of graphic systems. However, the object-oriented concepts it fostered did not gain wide acceptance. This may have been due the interactive capabilities overshadowing these ideas, or perhaps the object-oriented model itself was not complete enough [Rentsch,1982:55]. In either case, the object-oriented concepts that did survive were confined to the portrayal of graphic images on the screen, defining graphic images in terms of variables, and duplicating images via instantiation.

The majority of the graphic systems in use today (GKS, CORE, PHIGS, etc) have been implemented within traditional programming enviroments. FORTRAN, because of its early use, seems prevalent as the implementation choice, however 'C' [Denny,1986] and Ada [Hanson,1986] bindings have also become available. These systems embrace only the object-oriented concepts promoted by Sketchpad and little more. The idea of combining data with procedures is still not implemented.

It hasn't been since the development of true object-oriented systems, such as Smalltalk, that graphic systems have finally embraced the concepts associated with object-oriented systems. Several object-oriented graphical implementations are described in current literature [Goldberg and Robson,1983], [Wisskirchen,1986], [Lubinski and Hutzel,1984], and [Reiss,1986].

Support Environment. The graphical object-oriented environment implemented as part of this thesis, provides a flexible, yet consistent framework for defining a wide range of graphic metaphors. This section describes these metaphors without providing the detailed implementation concerns. The reader should refer to Appendix C for implementation details.

Currently three views of graphics programming are supported. They are primitive, user interface, and application views. The primitive view provides the foundation of the system. It supports the essential, inseparable graphic constructs that allows other views to be built. The user interface provides the interactive mechanism for the environment. It is through this level that the user can directly manipulate the system. The application view ties together both the primitive and user interface views to satisfy a user's requirement. It is the most dynamic portion of the environment,

allowing multiple applications to be constructed from primitive and user interface components.

Each view provides a different level of development abstraction, yet they are handled by environment in the same, consistent manner. The user is never forced to jump abstraction levels when developing software. This allows the user to mix abstraction levels freely without being burdened with the implementation details. A consistent abstraction is maintained by making all components in the environment objects. All components of the environment are accessed via a consistent message passing schema, regardless of the abstraction level they represent.

Primitive View. Primitives form the basic foundation of all graphic environments. They are the simplest objects in the environment serving as building blocks for more complex objects.

Currently six classes of primitives are supported; namely, Circle, Line, Point, Polygon, Rectangle, and Text. Each class defines a unique geometric figure and methods that are unique to its form. All graphic images displayed on the screen are combinations of one or more of these objects.

All primitive classes inherit their attributes from a Graphics Primitive class. This class gives each primitive, characteristics common to all graphic objects. Specifically, the Graphics Primitive class provides the following attributes:

cx,cy:	Center location of the object.
color:	Defines the objects color.
solid fill:	Indicates if the object is solid or not.
draw mode:	Indicates how the object is drawn on the screen.
area:	The objects area on the screen (in pixel units).
extent:	Defines the objects rectangular extent.

In addition to the classes defined above, a composite class is also supported. The class, Composite, provides instances that are composed of instances of other primitive classes. In other words, the Composite class provides the capability of constructing complex graphic images from simple graphic primitives, yet the resulting graphic image is treated as a single entity. For example, one could instantiate an object call Aircraft. Aircraft would refer to a single entity, but in actuality Aircraft is a composite of more general parts (i.e. objects) such as Fuselage, Wing, Tail, etc. Together all the parts define an aircraft.

Composite objects may also contain other composite objects. For example, the object 'Wing' could actually be a composite object consisting of more simpler objects such as Aileron and Flap. Together these simple objects define Wing, which in turn is used in the definition of Aircraft. This nesting of composite objects is very similar to graphic structures implemented in graphic modeling environments such as PHIGS [Abi-Ezzi and Bunshaft, 1986]. The reader should refer to Appendix D for a detailed description of the composite and graphic primitive classes.

User Interface. Currently, there are seven user interface classes supported by the environment. These classes are Command, Cursor, Dialogue Window, Display Window, Option Window, and Pop-Up Menu. Each class provides a unique way of allowing the user to interface to the environement. A functional description of each class follows.

Command:	Supports the display and selection of commands via a command button.
Cursor:	Handles the form and visibility of the cursor.
Dialogue Window:	Provides a consistent means of querying the user.

Display Window:	Provides an interactive medium for display and manipulation of primitve and composite objects.
Mouse:	Handles all mouse functions, such as, tracking and event queuing.
Option Window:	Provides a standard way of presenting objects for selection.
Pop-Up Menu:	Dyanmic medium for command selection.

A detailed description of each user interface class is provided in Appendix D.

Application. The final area supported by the environment is the application. An application is the actual task or requirement that the user is attempting to satisfy. Applications are supported by assembling together various components from the primitive and user interface classes. These classes should support all the components needed to build an application. If they don't, a new class should be constructed, rather than implementing the capability within the application itself. This will not only expand the number of classes available for application development, but will also provide consistency within the environment. Software development becomes the process of creating classes instead of programs.

Applications are viewed as master objects controlling the interaction of subordinate objects. They do not impart any new capabilities to the environment, rather they simply rearrange existing capabilities (classes) to perform new tasks. This is similar to a 'tinker-toy' set. The components of the set constitute the environment; how they are put together determines the application. This same metaphor can also be applied to the cockpit design process.

The environment was originally intended to support three applications, Layout Editor, Symbol Editor, and the Librarian. But because of its generic nature, it should be possible to support a multitude of different applications.

Application

The ALC prototype consists of four separate but interrelated components, namely; the user interface, Layout Editor, Symbol Editor, and the Librarian. Implementation of all four components was beyond the scope of this thesis effort. As such, a concerted effort was made to demonstrate the feasibility of the ALC concept by implementing the user interface and Layout Editor.

It was felt that the exclusion of the Symbol Editor and the Librarian would not significantly impact the primary goal of the ALC prototype, which was to demonstrate the feasibility of interactive cockpit display generation. Those capabilities supported by the Symbol Editor and Librarian (i.e. symbol construction and archival) could be initially emulated within the Layout Editor, with the understanding that a complete implementation of the ALC prototype would entail a detailed implementation of the Symbol Editor and Librarian. With this in mind, the remainder of this chapter will focus on presenting the implementation of the user interface and the Layout Editor.

User Interface. The user interface was implemented more as a part of the support environment than as a part of the ALC prototype itself. The functions and capabilities of the user interface tend to be more generic in

nature than application specific. The user interface is more of a 'kernel' than a stand-alone application.

Layout Editor. The purpose of the Layout Editor is to support the creation and subsequent modification of cockpit layout designs. The Layout Editor currently supports three cockpit layout types, aircraft ordnance loading, basic HUD design, and threat situation displays. These types were chosen because they represent a diverse range of possible applications to be supported by the ALC. The Layout Editor is currently limited to three layout types due to the lack of librarian support. Because the Librarian is not implemented, it became necessary for the Layout Editor to emulate a primitive librarian system. The primitive library is intended only for demonstration purposes, it is not intended to be a fully functional library. When the Librarian is finally implemented and integrated with the Layout Editor, the Layout Editor should be able to support an unlimited number of cockpit layout types.

Besides supporting only a limited number of layout types, the number of symbols defined for each type is also limited. Each symbol associated with a type must be hardcoded to that type. If the user wishes to add, delete, or modify a symbol, an off-line change to the software definition of the symbol is required. Again, this is directly related to the lack of a symbol editor. Once a symbol editor is implemented, it should be possible to modify a cockpit symbol while on-line, for any of the layout types in the library.

Although the Layout Editor suffers from the direct lack of support of the Symbol Editor and Librarian, the minimal capabilities these components provide, are emulated within the Layout Editor. The Layout Editor presents a complete picture, in and of itself, even without the implementation of the

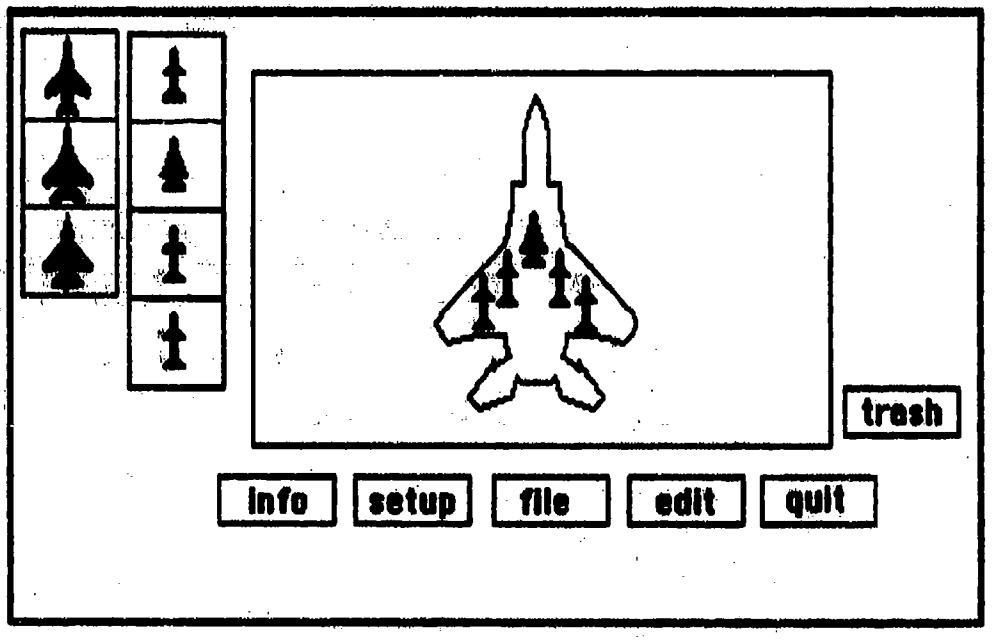

Figure 13. Example of an Aircraft Ordnance Loading Layout.

Symbol Editor and Librarian. The Layout Editor is best illustrated by examples of the actual layout types suppported.

Aircraft Ordnance Loading. The first layout type supported by the Layout Editor provides the designer with the capability of configuring different types of aircraft with different types of ordnance. Currently two types of symbol classes compose this layout type; namely, aircraft silhouettes and missiles. The aircraft silhouette class currently contains the figures of three aircraft; F-4, F-15, and the F-16. The missile class is composed of four different types; Harm, Maverick, Sidewinder, and Sparrow.

With these two symbol classes, the designer can construct aircraft loading displays similar to Figure 13. Figure 13 illustrates an F-15 configured with four Sparrow and one Maverick missile.

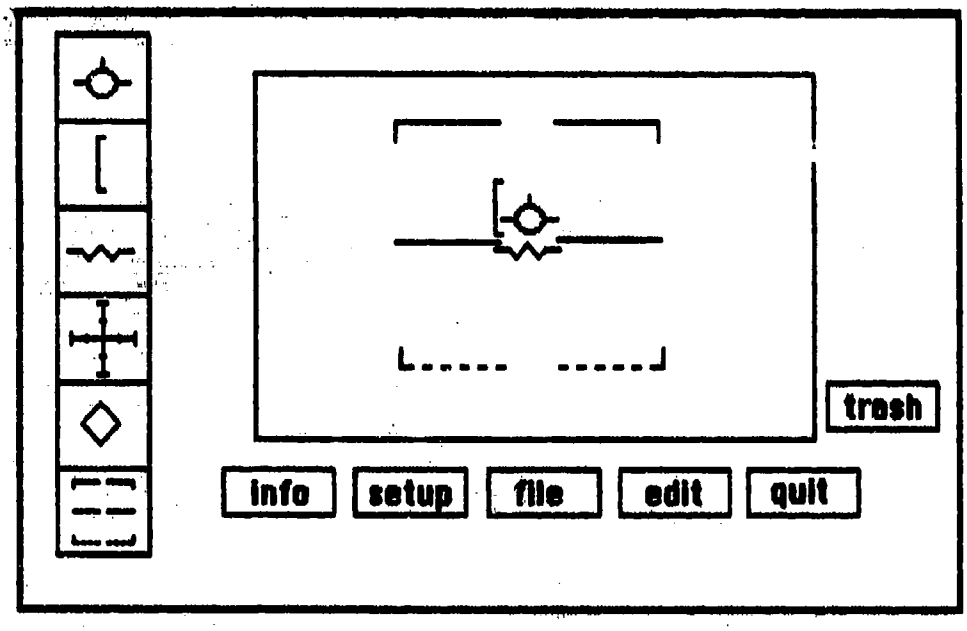

Figure 14. Example of a Simple HUD Layout.

HUD Design. The Layout Editor currently supports a primitive HUD design capability. Only a single symbol class is provided for component selection. This class contains the following symbols; flight path marker, angle of attack error indicator (aoa), aircraft reference symbol, inertial landing system (ils) bars, missile aiming rectile, and pitch ladder. Figure 14 shows a possible HUD created from this class. The HUD in Figure 14 contains a flight path marker, aoa error indicator, aircraft reference symbol, and pitch ladder. When the Symbol Editor and Librarian are implemented, it should be possible to have individual symbol classes for each of the symbols shown in the current class. This would provide the designer with an interactive means of experimenting with different HUD representations.

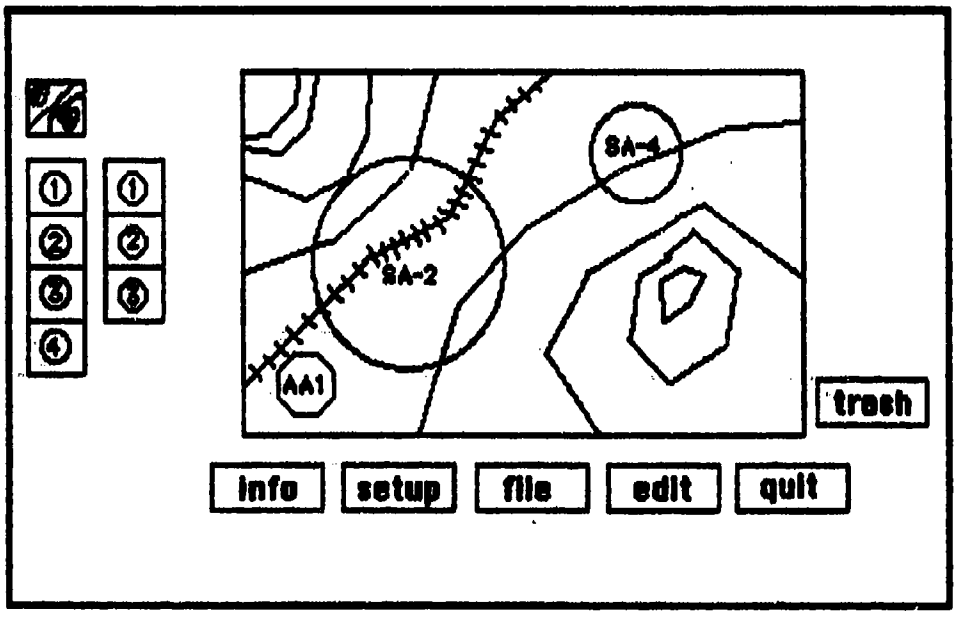

Figure 15. Example of a Threat Situation Layout.

Threat Situation. The last type of layout supported by the Layout Editor is the threat situation layout. Threat situation provides the pilot with a 'birds eye' view of the ground threats for a particular region. From such a display the pilot can quickly identify potential hazards and plan evasive actions.

The threat situation layout supports three symbol classes; terrain, surface to air missiles (SAM), and anti-aircraft (AA). The terrain class provides a single map for selection. The SAM class represents threats as circles. The size of the circle displayed depends upon the threat range of the SAM selected. Inscribed within each threat circle is a number indicating the SAM type. The same representation holds for AA, except AA threats are represented as octagons.

Figure 15. illustrates an example of a threat situation display. This display contains two SAM sites and one AA site.

It should be reemphasized that the current limitation of three layout types is solely a result of not implementing the Symbol Editor and Librarian. More layout types could have been added to the Layout Editor, but is was felt that the current layout types justly demonstrate the Layout Editors capabilities.

Summary

This chapter has presented the implementation of the ALC prototype. The prototype was implemented in three phases; hardware, support environment, and the actual application.

Hardware implementation dealt with defining the hardware environment that the ALC prototype was hosted on. It is currently supported by a Raster Technologies Model One/25 graphics processor.

A graphical object-oriented environment was implemented as a part of this thesis to provide a flexible foundation for the actual implementation. Object-oriented concepts, derived primarily from Smalltalk, shaped that implementation.

Implementation of the ALC prototype was originally targeted to include the user interface, Layout Editor, Symbol Editor, and the Librarian. Such and effort was soon discovered to be beyond the scope of a single thesis. Because of this, a functional subset of the ALC prototype was chosen for implementation. This subset, consisting of the user interface and Layout Editor, encompasses the essence of the ALC prototype effort by providing the capability for constructing cockpit layout designs. The Layout Editor currently supports three type of layouts; aircraft ordnance loading, simple HUD, and threat situation. Examples of these layouts were given. The

Layout Editor is capable of supporting other layouts, but it was felt that these selected layout types were diverse enough to demonstrate the generic editing capabilities of the Layout Editor without adding additional layout types.

VIII. Conclusions and Recommendations

Conclusions

This thesis has been a preliminary attempt to define (quantify) a possible ALC representation for the AC/CSRL. It has focused on the rapid prototyping of pictorial type cockpit displays via the use of existing cockpit layouts and symbology. This research has resulted in the design and implementation of a highly interactive, graphics based environment known as the ALC prototype.

The ALC prototype was designed to support four major AC/CSRL ALC requirements; namely,

1. user supportiveness,
2. interactive creation and modification of cockpit layouts,
3. interactive creation and modification of cockpit symbology, and
4. storage and retrieval of cockpit layouts and symbology.

Due to limited time and resources, the current ALC prototype implementation only supports the first two requirements; user supportiveness and cockpit layout generation. However, this implementation has shown to be sufficient to demonstrate the central theme of the AC/CSRL ALC concept; namely, the interactive support of cockpit display generation. Implementation of the remaining requirements would greatly enhance the capabilities of the ALC prototype and would provide a suitable framework for demonstrating a broad range of design and implementation issues associated the AC/CSRL ALC effort.

This thesis has demonstrated the concepts proposed by the AC/CSRL effort are feasible and can be implemented using todays technology. What this thesis did not demonstrate, nor did it attempt to, was what the final ALC representation should be. Rather, this thesis proposed a candidate ALC representation based on several key ALC requirements.

Aside from demonstrating the ALC concept from the user's viewpoint, this thesis also addressed the use of object-oriented concepts in the design and implementation of the ALC prototype software. A consistent object-oriented metaphor was applied to all levels of the ALC effort. At the user level, all interactions were viewed as the manipulation of cockpit 'objects'. The software design was based on an object-oriented design methodology, and the actual software itself was implemented in an object-oriented fashion. This approach provided a consistent framework from the application down to the implementation; this is seldom attainable with more traditional approaches.

In addition to providing a consistent metaphor, an object-oriented implementation supporting multiple inheritance, proved a viable means of rapidly generating software. Classes, once fully implemented and tested, served as the foundation for building other classes. Building new classes upon the functional capabilities of existing classes reduced the need for extensive software development and testing.

Although this thesis addressed a specific application, the rapid prototyping of pictorial cockpit displays, the graphics environment that supports the prototype ALC was developed as generic as possible. It should be possible to extend the ideas and concepts embodied in this environment to other areas requiring the rapid prototyping of pictorial displays.

Recommendations

Recommendations are divided into two areas; short term and long term. Short term recommendations suggest ways for immediately enhancing the capability of the ALC prototype. Long term recommendations are enhancements or issues identified as part of this thesis effort that could impact directly the AC/CSRL development effort.

Short Term. The most immediate short term enhancement that could be applied to the existing ALC prototype implementation would be the implementation and integration of the Symbol Editor and Librarian with the Layout Editor. Currently, the Layout Editor must emulate specific capabilities originally intended to be supported by the Symbol Editor and Librarian (i.e. cockpit symbol construction and layout retrieval). The emulation is primitive and detracts from the original intention of the Layout Editor, namely; cockpit layout construction. By implementing the Symbol Editor and Librarian, the ALC prototype would truly be a supportive environment.

The ALC prototype could also be enhanced by rehosting the software on a dedicated graphics workstation. Currently, the ALC prototype software is hosted on a Raster Technologies Model One/25 graphics processor connected to a time-shared VAX 11/785. In this configuration, the Model One/25 serves as an intelligent graphics terminal, while the VAX 11/785 performs the majority of the computational tasks. User interaction is often hampered by this set up. Besides being constrained by a relatively slow (9600 baud) communication channel, the VAX is sometimes heavily utilized by other applications resulting in intermittent bursts of high user response followed

by no response at all. This hit and miss response mode significantly detracts from the interactive capabilities of the ALC prototype. A dedicated graphics workstation should eliminate this problem providing the user with a more reponsive system.

Display representations could also be enhanced by providing a three dimensional extension. Currently only two dimensional design representations are supported. A three dimensional extension would broaden the scope of possible design representations that could be supported by such an environment.

As more and more military software systems are being implemented in Ada, it might be warranted to translate the current implementation to Ada to provide a better integration with other software packages. Although Ada software can be designed using an object-oriented approach [Booch,1986], the object-oriented nature of the design is lost in implementation. Ada does not directly support an object-oriented implementation schema such as Smalltalk, or the environment implemented in this thesis. Recent efforts to provide an object-oriented framework in Ada has met with some degree of success, yet a complete object-oriented implementation of Ada looks doubtful [Braaten and Hanson,1986]. As such, some of the object oriented nature of the ALC prototype implementations will have to be compromised for an Ada implementation.

Long Term. The long term recommendations presented here pertain more to the AC/CSRL project as a whole, rather than individual enhancements to the ALC prototype. The recommendations identified during this thesis effort should be considered as possible requirements for the AC/CSRL's ALC.

When testing the layout capability of the Layout Editor, it was often noticed that different layouts could be constructed in which the layout representation could not be matched by its corresponding implementation in the real world. For example, the aircraft ordnance loading layout (Figure 13) allows aircraft to be configured with different type of missiles. In a real aircraft cockpit, this display type would actually represent the aircrafts' current ordnance status. As such, there is fixed physical limit to the amount and type of ordnance that a specific aircraft can deploy. The ALC implemented as part of the AC/CSRL should provide some means of determining if a cockpit layout is valid. Design verification could be achieved by integrating a knowledge base with the graphics editor used.

Another feature not implemented as part of the ALC prototype, which should be incorporated in the AC/CSRLs' ALC, is the capability to describe object behavior. It is not enough just to be able to describe an object's form, the designer should also be able to, from the ALC, describe an object's behavior. The definition of an objects behavior could include spatial contraints (i.e. vertical or horizontal movement) and identification of which responses from the intended environment invoke a reply from the object. From such definitions, it should be possible to generate application software to dynamically model the layout. Related work in this area is currently being conducted by [Foley and McMath,1986] and [Hollan et. al.,1984].

Appendix A: Design Methodologies

Software design can be viewed as a decomposition process guided by an abstraction criteria. The decomposition process divides the original problem space into a series of smaller, simpler problem spaces while abstraction guides the process by imposing restrictions on how the problem space is divided. The choice of the abstraction criteria directly impacts the structure of the final design.

Traditionally two forms of abstraction criteria have been applied to the decomposition process: functional (process-driven) and data-structure (data-driven). Functional decomposition techniques have been popularized by practices known as top-down design, Structured Design [Constantine and Yourdon,1979], and step-wise refinement [Wirth,1971]. These techniques approach decomposition based on an algorithmic or functional view of the problem domain [Booch,1986:211]. Large, complex problems are divided into a series of smaller more managable subproblems. These subproblems are solved or further decomposed into a series of even smaller subproblems. Decompositon is repeated until the entire problem is stated in terms of smaller, solvable subproblems. The subproblems solutions are then combined to solve the original problem.

The program structure derived from functional decomposition embodies a hierarchical refinement of functional detail. Each level of the hierarchy expresses an abstract definition of system functionality. The top most level defines 'what' the system should do. Succeeding levels refine this definition until the most fundamental operations are defined. Each fundamental operation specifies 'how' a particular function in the system performs. By

combining these operations under a hierarchical network of control flow a hierarchical program structure of functional components is created.

A second approach to software design decomposes the problem space based on a data-structure abstraction. Methods developed by Jackson [Jackson,1983] and Warnier [Warnier,1977] are the most popular. These methodologies subscribe to the idea that the "structure of a software system should reflect the structure of the data processed by that system" [Sommerville,1985:69]. Instead of decomposing the problem space based on how the system functions (i.e. functional decomposition), the problem space is decomposed based on an analysis of the input and output of the system data. This decomposition results in a hierarchical definition of the data structures that reflect the data processed by the system. The program structure is formated by transforming the hierarchy of data structures into a hierarchy of corresponding program units that process the data.

Traditional software design methodologies provide for the formulation of problem domain representations (designs) based on either a functional or data-structure viewpoint. Functional decomposition techniques have concentrated on defining the operations in the problem domain with little regard to the data structures needed. On the other hand, data-structure decomposition techniques have taken just the opposite approach. The data-structures are defined first; functions are defined as an afterthought to use the structures. Program structures generated by these techniques seldom portray a clear and direct representation of the problem domain. The program structure ends up representing a set of operators (functions) or a set of operands (data) [Cox,1984:58]. A synthesis of function and data is absent. This causes the program structure to be a transformation of the problem domain rather than a direct mapping of it. The program structure

becomes "removed from the problem space" [Booch,1983:40].

An alternative to the more traditional software design methodologies (i.e. functional and data-structure) is based on an object-oriented approach. Object-Oriented Design (OOD) is a software design technique in which "decomposition is based upon the concept of an object" [Booch,1986:211]. Objects encapsulate both the state (data-structure) and behavior (function) of entities in the problem domain [Cox,1984:57]. A synthesis of data-structure and functionality is achieved.

The idea of combining data and function as a single decomposition criteria is rooted in the principle of 'information hiding' [Parnas,1972]. Information hiding conceals the internal processing details of individual levels of the program structure from each other. Each level has access only to information that is pertinent to its processing needs. Access to information from other levels is prohibited. Each level encapsulates its own data structure and operations. Levels communication through well-defined interfaces. Knowledge about a levels' internal processing details are hidden from the calling level, only the interface syntax is visible.

The value of OOD arises from information hiding. Objects are abstract entities containing both state (data) and behavior (operations). Every object hides its internal details from other objects and communicates via message passing (well-defined interfaces). Objects provide an abstraction medium for consolidating the ideas of information hiding. Program structures that were once transformations of the problem domain (i.e. sets of operands or sets of operators) are now composed of objects (operands and operators). Decomposition is viewed as an identification process. Objects are identified in the problem domain and mapped directly into the program structure. Functionality and data-structure are no longer viewed as separate

attributes of the problem domain. "As a result, the designer is not forced to restate his problem in computer-domain terms, where everything must be either an operator or an operand" [Cox,1984:58], but rather defines the design in terms that exist in the problem domain.

Basically, OOD can be generalized in the following four steps:

1) Definition, examination of the problem domain,

2) identification of the objects in the problem domain,

3) identification of operations performed on the objects, and

4) implementation of the objects.

[Booch,1986:213; Buzzard et. al.,1985:11; Cox,1984:58]

The first step, problem definition, is common to all design methodologies. Its goal is to define a complete and understandable description of the problem domain.

The second and third steps in the OOD process involve the identification of objects and associated operations. The procedure for doing this seems more of an art than a science. Depending on the complexity of the problem domain, the task of object and operation identification may be intuitively obvious or seemingly impossible.

Attempts to formalize the identification process have been popularized by methods proposed by Abbott [Abbott, 1983] and Booch [Booch,1983]. Their strategy is based on identifying objects and associated operations by extracting noun and verb constructs from a natural language description of the problem. Objects are associated with noun, pronouns, and noun clauses, while operations are associated with verbs, verb phrases, and predicates [EVB, 1985: 2-6,2-9]. Proponents of the strategy claimed to have used this technique successfully for small to medium sized (up to 30,000 lines of code) programs [EVB,1985:1-2]. Yet skepticism remains about the

applicability of such a method for large complex programs and whether a natural language description can produce a clear and concise enough description of the problem domain for this technique to be used [Sommerville,1985: 94].

The final step, implementation, performs the actual mapping of the design into software. The degree to which the software retains its object-oriented structure depends directly on the language used for implementation.

Appendix B: Characteristic and Terminology Associated with Object-Oriented Systems

This appendix provides an introduction to the characteristics and terminology associated with object-oriented systems. This appendix is not intended to be a tutorial on the subject, but rather a consolidation of terms and characteristics found throughout this thesis. Specifically, this appendix addresses three main concepts associated with object-oriented systems; namely, class, message passing, and inheritance.

Classes

Objects are the sole inhabitants of an object-oriented environment. They encapsulate the properties of data (operands) and procedures (operators) into a cohesive whole. The data, or *instance variables*, define the instrinsic properties of the object. For example, a line object may contain instance variables that describe its form as two end points, starting_end_point and terminating_end_point. The instance variables defined for an object are only known to that object.

Objects also contain procedures or *methods*. They are the sole means of manipulating an object's instance variables. Only those methods defined for an object can manipulate that object's instance variables. Methods defined in other objects are forbidden from directly modifying instance variables of different objects.

Objects are implemented as instances of *classes*. Classes serve as a 'blueprint' for constructing all objects in the system. All objects are an

instance of a single class; however, classes can have multiple instances (i.e. objects). For example, the class Rectangle defines all the information necessary for creating a rectangle. Instances of the class Rectangle would define a set of different size rectangles. Each rectangle has in common the property of 'rectangleness', yet they differ in their presentation (i.e. size). Classes capture the 'gestalt-ness' of the instances.

Classes consist of class variables and storage for the instance methods. Class variables differ from instance variables, in that, class variables are shared by all the instances of the class. In contrast, each instance maintains its own instance variables and has exclusive access to them.

Besides class variables, each class maintains the actual implementation of the instance methods. In principle, every instance of a class could maintain a personal copy of the methods; however, this strategy is wasteful (memory) and serves no useful purpose. By confining the actual implementation of the methods to the class definition, memory requirements are minimized since all instances share the same methods. Viewed in this way, a class can be thought of as collections of objects that have the same operations in common [Curry and Ayers, 1984:520].

Message Passing

Objects communicate via a message passing paradigm. Instead of directly invoking a procedure (operator) to perform an operation on an object, one sends a message to that object. The receiving object determines how to handle the message. The receiver has three options available to it;

1) it can invoke one of its methods to implement the message,

2) it can ignore the message, or

3) it can pass the message on to another object.

In either case, the sender has no control over 'how' or 'who' finally processes the message. The sender blindly trusts that the receiver will do 'the right thing' [Rentsch,1982:54]. The messages that an object responds to defines the object's interface to the rest of the system.

The principle that makes message passing possible is binding. Binding is the act of translating the application software into actual machine addresses for execution. There are two basic forms of binding, static and dynamic [Aho and Ullman,1979:37]. Static binding is usually performed at compile time. The software is bound to actual machine address prior to execution. This method is very efficient, binding all functions/procedures to a specific data type. It becomes impossible to use a procedure to manipulate integers on one line then use that same procedure to manipulate strings on the next line. The procedure must be explicitly defined with an appropriate data type and used with that data type consistently throughout the software. On the other hand, dynamic binding delays the type checking until run time. It becomes the responsibility of the software environment to determine how to handle multiple data types, making it possible for a procedure to manipulate different data types.

Object-oriented systems use some form of dynamic binding to support message passing. Messages are sent to objects to elicit a desired action. Since the message content is not checked until run-time, multiple objects can be sent the same message. It then becomes the responsibility of the receiving object to determine how to interpret the message. For example, the message 'draw' would elicit a different response from a Line object than

it would from a Circle object. Both objects would interpret the message and perform the appropriate draw response. The meaning and syntax of the message is the same for both, but the means in which the message is implemented is dependent upon the object that receives it. The reponsibility of implementing the message rests squarely with the receiving object.

Inheritance

Object-oriented systems also support a concept known as inheritance. Inheritance is a means of creating specializations of existing classes. The new class, known as the *subclass*, inherits all the class variables, instance variables, and methods of the existing class, or *superclass*. The subclass is distinguished from its superclasss by unique class variables, instance variables, and methods. Methods defined in the subclass may override the methods inherited by the superclass or can be used to enhance the superclass methods. [Pascoe,1986:142]

There are two basic forms of inheritance, hierarchical and multiple [Stefik and Bobrow,1986:46-49]. Hierarchical inheritance is the simplest of the two. It restricts the number of classes that a subclass may inherit to one. Each subclass in the hierarchical scheme has only one superclass. This results in an inheritence structure similar to a tree in which each node is the decendent of only one previous node. Multiple inheritance, one the other hand, allows multiple classes to be inherited by a single subclass. This results in a tree structure in which a node can be descendant from one or more other nodes. Figure 16 illustrates this difference.

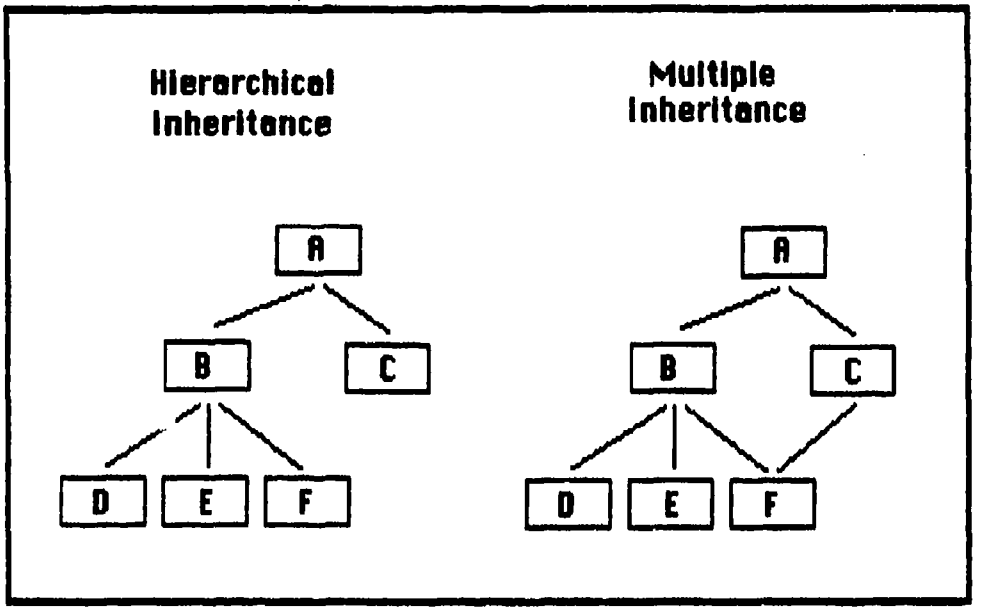

Figure 16. Example of Hierarchical and Multiple Inheritance.

When viewing inheritance diagrams, it is important to understand which direction the inheritance follows. Upon initial inspection, one would assume that class A, would inherit characteristics from both B and C. In inheritance diagrams, such as Figure 16, this is just the opposite. It is classes B and C that inherit characteristics from A. In the hierarchical example, each class inherits characteristics from only one other class. In the multiple inheritance example, class F inherits characteristics from both B and C.

The choice of which inheritance mechanism is used will often depend on the class structure of the application. For applications where classes are mostly independent of each other, a hierarchical inheritance structure could suffice. However, in applications, where classes are highly interrelated, the use of a multiple inheritance structure is warranted. Multiple inheritance structures provide an extra level of flexibility not usually associated with strictly hierarchical structures. Since classes in a

multiple inheritance structure can inherit multiple classes, adding new classes is simply a matter of establishing new inheritance links to the existing structure. Adding new classes to a hierarchical stucture could entail a readjustment of the entire structure.

Appendix C: An Object-Oriented Extension of the 'C' Language

This appendix describes an object-oriented extension of the 'C' language that was implemented as part of this thesis. It differs from previous object-oriented extensions of 'C', [Stroustrup,1983] and [Cox,1983], in that it does not require the code to be pre-compiled. All extensions are implemented in standard 'C'. This should make this implementation transportable to other 'C' systems.

Three primary extensions were added, namely; class type, message passing, and multiple inheritance. The models for these extensions were the Smalltalk language [Goldberg and Robson,1983] and Traits [Curry and Ayers,1984]. A description of these extensions is provided in the following sections.

Classes

The class concept is implemented as a data structure consisting of three interrelated components; object, class, and method table. Each component is implemented with an identical 'C' structure (Figure 17). Each node in the structure consists of six fields. The first field, **type**, identifies the component for which the node is being used. The second field, **ptr_type**, provides a generic means of assigning the different components to the node. Field three, **super**, supports the concept of inheritence by serving as a link to other class nodes. The fourth field, **path**, is used to link the different components together. The fifth and sixth fields, **next** and **back**, are used by the system to construct component lists of object and class types.

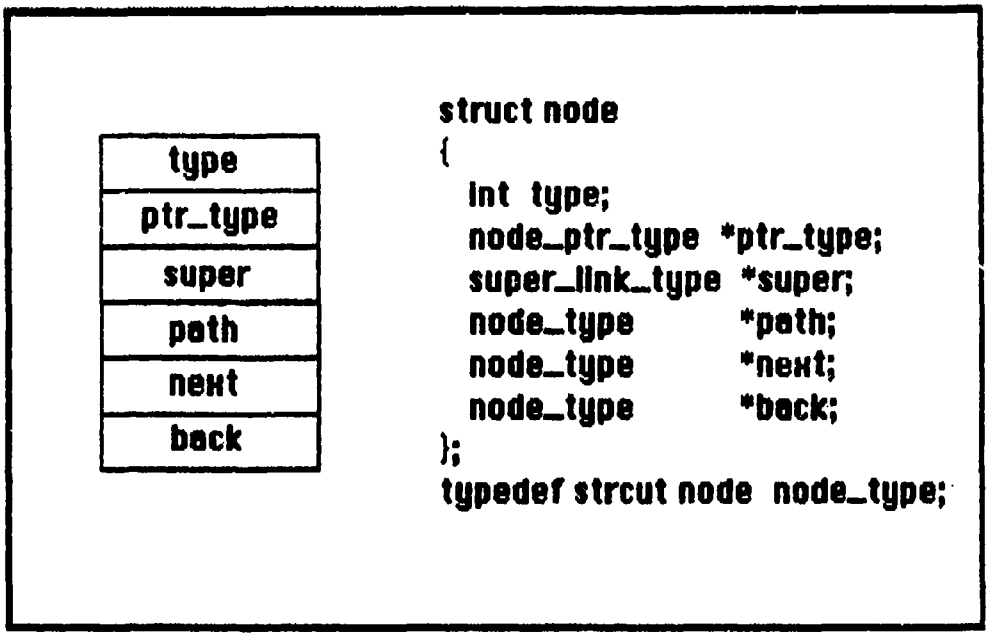

Figure 17. Generic Node Structure.

Object Node: An object node represents an actual instance of a class. Multiple object nodes can be instantiated for a single class. Each object node instantiated shares all the class variables and methods defined for the class. However, the instance variables associated with an object node are private to that node.

Objects are instantiated dynamically at run-time. The application program will typically send a message to a class requesting the creation of an object. The class, in turn, will invoke the appropriate class method to allocate storage from the memory heap for the object. The new object is then linked to the class node. A handle, or pointer, is returned to the application identifying the new object. Objects can also be deleted from the system in opposite manner by nulling its pointers and deallocating its storage from memory.

Class Node: Class nodes are the foundation of the data structure. All object nodes are linked to class nodes. There exists a many to one mapping between object nodes and class node. Only one class node is associated with an object nodes. Class nodes provide the common channel for all method invocation.

All classes are instantiated are run-time by the function setup(). Setup() allocates storage for each class node and assigns global pointers for each class. These pointers are used by applications to identify the class in which they want an instance. Setup() should be the first function called by the application and it should be called only once.

Method Table Node: The method table node provides access to the methods associated with a class. Methods are maintained in a linked list structure. Each node in the list contains a selector; to identify the method, a parameter count, and the actual machine address of the 'C' function that implements the method. Figure 18 illustrates the linked list structure of the method table. The method table node and list structure is instantiated automatically when the class node is instantiated.

The inter-relationship of object, class, and method table nodes illustrated in Figure 19 represents the system structure containing a single class. As more classes are added, the system structure takes on the form of a tree (Figure 20) where each node in the tree represents the structure as presented in Figure 19. The branches in the tree form the inheritance chain between classes. All inheritance in the tree terminates in a single node or class. This class is commonly refered to as Object. The class Object is the only class in the system that does not have a superclass.

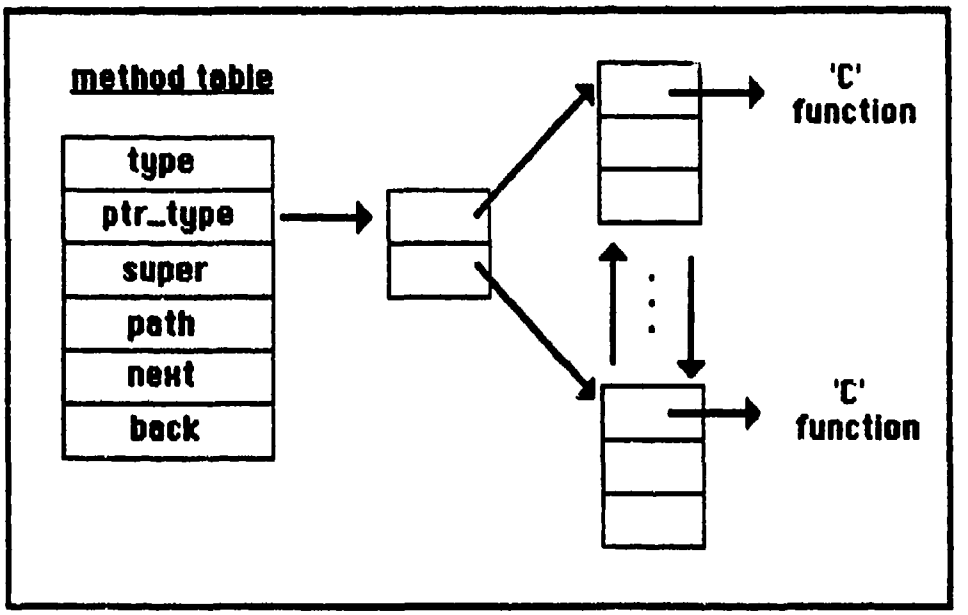

Figure 18. Method Table Structure.

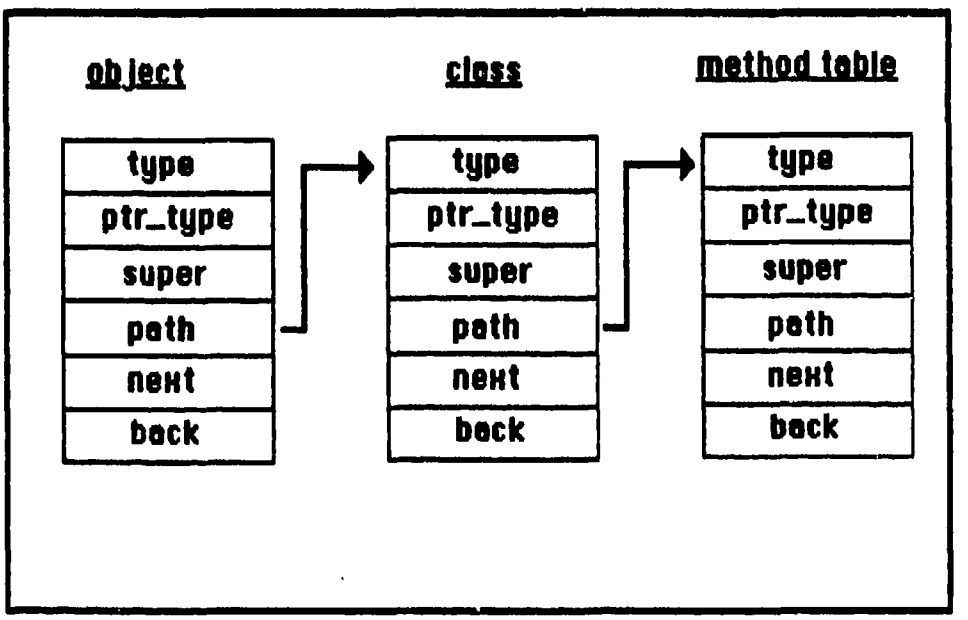

Figure 19. Interrelationship Between Nodes.

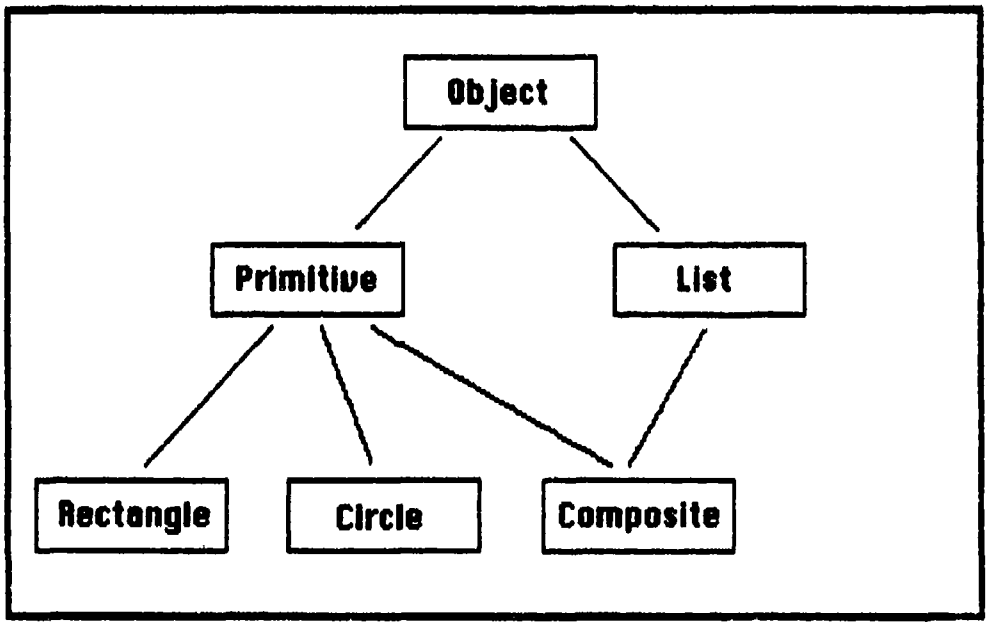

Figure 20. Structure of Multiple Classes.

Message Passing

All objects interact via message passing. Essentially, message passing is a form of indirect function invocation. Instead of directly invoking a function, as is done in Ada or Pascal, a message is sent to an object for processing. The receiving object then determines how the message is handled, not the sender.

Message passing is performed via two 'C' functions, msg() and broadcast_to_super(). Msg() is the primary way of sending messages to an object. Msg() supports message passing directly to a different object or allows a message to be sent to itself. Broadcast_to_super() allows a message to be sent immediately to the objects superclass by bypassing the object's methods.

Both procedures employ the same parameter passing order or protocol. The protocol of each function consists of a receiver, the message (or selector), and any parameters associated with the message. The receiver is the object in which the message is intended for. The receiver may be a different object than the sender or the sender may send itself a message. In either case, the message must always be addressed to an object. The system currently performs little error checking. Any attempt to send a message to a non-existent object will most likely cause the application to crash.

The selector is the actual message text, character string, that indicates which method should be invoked by the receiving object. Each message can have up to five associated parameters. These parameters can be any of the following data types : char, int, float, and node_type.

Examples of message passing follows:

```
msg( circle,"setRadius",20);
```

This example illustrates a message being sent to the object 'circle'. The sending object has requested that 'circle' change its radius to 20. If the message had been:

```
msg( circle,"Radius",20);
```

Circle would have ignored it and retained it original radius value since 'circle' does not understand what "Radius" means. It thus becomes very important to send the correct format of the message to the object to ensure that the desired action is performed.

Msg() also has the capability of returning a response from an object. In such cases, the sender must know beforehand the format of the reply, for example, if sender requests the circle radius:

```
radius = (int)  msg( circle,"getRadius");
```

The sender must know in advance that the result will be of 'int' type. The sender should not expect something different. More complex formats can also be returned from msg(), for example:

```
new_circle = (node_type *)  msg(circle,"clone");
```

returns a pointer to a copy of the original 'circle' object.

Broadcast_to_super() does not provide a reply capability. Thus it should only be used for messages that generate no response.

Multiple Inheritance

Having described how message passing is performed on the conceptual level, it is now worthwhile to investigate how message passing is handled by the system. When an object receives a message, it scans its method table comparing the selector passed to it against selectors stored in the method table. If a match is made, the 'C' function associated with that entry is invoked. If no match is found, the object has one of two options. It can simply ignore the message and return control back to the sender, or it can pass the message on to its superclass.

An object can ignore a message only if it has no superclass to pass the message on to. Since the class Object is the only class with no superclass, it is the only class the is allowed to ignore a message.

Passing the message on to an objects superclass, cause the superclass to search its method table. If no match is found, the message is passed on to the superclasses' superclass, or the original receiving objects super-superclass. This recursive procedure continues until a match is found or until Object receives the message. If Object can match the message in its method table, the function is invoked, otherwise the message is ignored and control is return to the sending object.

The ability of an object to pass on a message to a superclass is the basis for inheritance. When a message is processed by the superclass, the original instance is said to inherit that supperclass's method. Even though the method that implemented the message is not defined in the objects method table, it can be used as if it were. When an object only inherits methods from a single superclass, inheritance is viewed as hierarchical. All classes in the inheritance chain, inherit the methods of only one superclass, and that superclass inherit methods from only one super-superclass, so on and so on until the Object class is reached.

Multiple inheritance allows a class to inherit methods from multiple superclasses. The superclasses can then inherit methods from multiple superclasses, so on and so on, until all inheritance terminates with the Object class.

A multiple inheritance schema was chosen for this implementation. To support multiple inheritance a precedence list was established for each object (the field 'super' in the object node points to this list). This list keeps track of all the superclasses associated with an object. When a

message is sent to an object, the object's method table is searched. If does not contain the method, the method table of the first superclass in the precedence list is searched. If the search fails, the method table of the first superclass in the current superclass precedence list is searched. This continues until the Object class is reached. The class Object is represented by a precendence list that points to NULL. When a NULL is encountered, control is return to the previous superclass and the next class in the precedence list is searched. This continues until a match is found in one of the superclass's method table or until the calling object is returned control.

This may seem a bit complicated at first, but as more and more classes are added to the system, the power and elegance of this schema to absorb them without modifications to the software structure becomes evident. New classes are simply assigned pointers to the superclasses in which they wish to establish an inheritance with. All methods associated with the superclass are made accessible to the new class. This provides a high degree of software reusability among the classes.

Appendix D: Class Descriptions

This appendix describes the interface for all currently implemented classes. Figure 21 illustrates the inheritance relationship among these classes. These classes are divided into three categories; 1) graphic primitives, 2) user interface, and 3) miscellaneous. The graphic primitives are classes that describe basic graphic entities such as rectangle, circle, and polygon. The user interface consists of classes that support the ALC prototype user interface. The miscellaneous classes provide support for the other two class categories.

Classes are described in terms of variables and methods. Specifically, each class provides the following information:

1. **Class Name**

2. **Superclass:** The superclass(s) identify those classes that are inherited by the current class. All class variable, instance variables, class methods, and instance methods defined for a superclass are inherited by the current class.

3. **Class Variables:** Defines those variables that are unique to the class and shared by all instances of that class.

4. **Class Methods:** The methods (operations) that are uniquely defined for the class. The 'C' interface for each method is provide.

5. **Instance Variables:** Defines those variables that are private to each instance.

6. **Instance Methods:** The methods that are available to an instance. The 'C' interface of each method is provided.

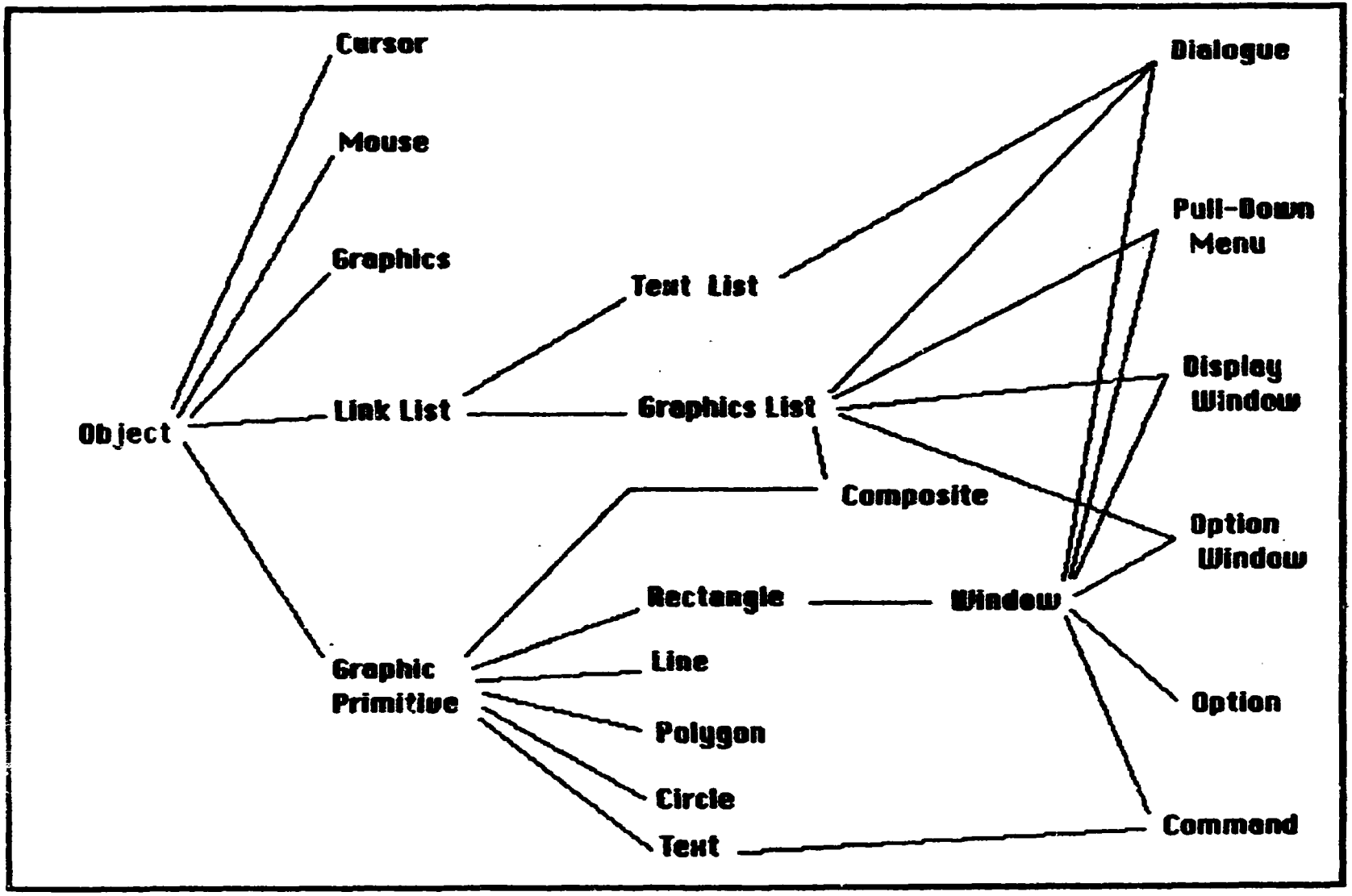

Figure 21. Inheritance Structure

Class: circle

Super Class: graphic primitive

Class Variables: none

Class Methods:

new
function: Instantiates a circle object.
circle = (node_type *) msg(circle_class,"new")

Instance Variables:

radius : defines the radius of the circle.

Instance Methods:

clone
function: Answers with a clone of an existing object.
object_clone = (node_type *) msg(circle,"clone")

setCircle
function: Defines where the circle is, and its' size.
msg(circle,"setCircle",[x],[y],[radius])

setRadius
function: Defines the size of the circle.
msg(circle,"setRadius",[radius])

scaleBy
function: Scales the circle in both the x and y directions.
msg(circle,"scaleBy",[factor])

draw
function: Draws the circle.
msg(circle,"draw")

internals

function: Prints the circles' instance variables.

msg(circle,"internals")

Class: graphic_primitive

Super Class: object

Class Variables: none

Class Methods:

new
function: instantiates a graphic primitive object.
primitive = (node_type *) msg(graphic_primitive_class,"new")

Instance Variables:

cx,cy : center point of graphics primitive
color : color of graphic primitive
solidFill : flag indication whether primitive should be solid filled
drawMode: determise how the image is drawn on the screen (i.e. normal or XOR)
area : area of graphic primitive in pixels
extent : the graphic primitives extent on the screen. The extent is defined as a rectangluar region.

Instance Methods:

clone
function: creates a clone of an existing object. Caller is returned a handle to a new object.
clone_object = (node_type *) msg(primitive,"clone")

setColor
function: sets the color of the graphic primitive.
msg(primitive,"setColor", [COLOR])

setSolidFill
function: sets whether a graphic object is displayed solid filled.
msg(primitive,"setSolidFill", [TRUE or FALSE])

setExtent
function: sets the rectangular extent of the graphic primitive. The area of the graphic primitive is automatically updated.
msg(primitive,"setExtent",[left], [bottom], [right], [top])

setDrawMode
function: Sets the drawing mode for the object.
msg(primitive,"setDrawMode",[mode])

getColor
function: Answers with the primitives color.
color = (int) msg(primitive,"getColor")

getSolidFill
funciton: Answers with the primitives solid fill status.
fill = (int) msg(primitive,"getSolidFill")

getExtent
funciton: Answers with the primitives extent.
extent = (extent_type *) msg(primitive,"getExtent")

getDrawMode
function: Answers wih the primitives drawing mode.
mode = (int) msg(primitive,"getDrawMode")

getArea
function: Answers with the primitives' area.
area = (int) msg(primitive,"getArea")

getCenterPoint
function: Answers with the primitives center point.
point = (point_type *) msg(primitive,"getCenterPoint")

containsPoint
function: Answers whether the primitive contains a specific point (TRUE or FALSE).
reply = (int) msg(primitive,"containsPoint",[x], [y])

moveTo

function: Moves (centers) the primitive over a specific point.

msg(primitive,"moveTo",[x],[y])

showExtent

function: Displays the primitives' extent.

msg(primitive,"showExtent")

hideExtent

function: Erases the primitives extent.

msg(primitive,"hideExtent")

internals

function: prints the instance variables of the primitive.

msg(primitive,"internals")

Class: line

Super Class: graphic primitive

Class Variables: none

Class Methods:

new
function: Instantiates a line object.
line = (node_type *) msg(line_class,"new")

Instance Variables:

start_x,start_y : the starting end point of the line.
end_x, end_y : the end point on the line.

Instance Methods:

clone
function: Answers with a clone of an existing object.
object_clone = (node_type *) msg(line,"clone")

setLine
function: Defines the end points of the line.
msg(line,"setLine",[start_x],[start_y],[end_x],[end_y])

scaleBy
function: Scales the line in both the x and y directions.
msg(line,"scaleBy",[factor])

draw
function: Draws the line.
msg(line,"draw")

internals
function: Prints the lines' instance variables.
msg(line,"internals")

Class: polygon

Super Class: graphic primitive

Class Variables: none

Class Methods:

new
function: Instantiates a polygon object.
poly = (node_type *) msg(poly_class,"new")

Instance Variables:

num_of_points : Number of points that define the polygon.

Instance Methods:

clone
function: Answers with a clone of an existing object.
object_clone = (node_type *) msg(poly,"clone")

addToPoly
function: Adds a point to the polygon
msg(poly,"addToPoly",[x],[y])

scaleBy
function: Scales the polygon in both the x and y directions.
msg(poly,"scaleBy",[x],[y])

draw
function: Draws the polygon.
msg(poly,"draw")

internals
function: Prints the polygons' instance variables.
msg(poly,"internals")

Class: rectangle

Super Class: graphic primitive

Class Variables: none

Class Methods:

new
function: Instantiates a rectangle object.
rect = (node_type *) msg(rect_class,"new")

Instance Variables:

left,bottom : lower left hand corner of the rectangle.
right, top : upper right hand corner of the rectagle.

Instance Methods:

clone
function: Answers with a clone of existing object.
object_clone = (node_type *) msg(rect,"clone")

setRect
function: Set the dimensions of the rectangle.
msg(rect,"setRect",[left],[bottom],[right],[top])

scaleBy
function: Scales the rectangle in both the x and y directions.
msg(rect,"scaleBy",[factor])

draw
function: Draws the rectangle.
msg(rect,"draw")

internals
function: Prints the rectangle's instance variables.
msg(rect,"internals")

Class: text

Super Class: graphic_primitive

Class Variables: none

Class Methods:

new
function: instantiates a text object
text = (node_type *) msg(text_class,"new")

Instance Variables:

length	:	Text length in pixels.
height	:	Text height in pixels.
size	:	Font size.
font	:	Style of font.
text_string	:	Actual character string.

Instance Methods:

clone
function: Creates a clone of an existing object. Answers with a handle of the new object.
clone_object = (node_type *) msg(text,"clone")

setText
function: Defines text as a string of characters.
msg(text,"setText", [string])

getTextSize
function: Answers with the current size of text.
size = (int) msg(text,"getTextSize")

scaleBy
function: Scales the text object in both the x and y directions.
msg(text,"scaleBy", [factor])

startWriteAt

function: Defines the location where the lower left hand corner of the text should begin.

msg(text,"startWriteAt",[x], [y])

draw

function: Draws the text string.

msg(text,"draw")

internals

function: Prints the instance variables of the text object.

msg(text,"internals")

Class: commands

Super Class: window
text

Class Variables: none

Class Methods:

new
function: Instantiates a command object.
command = (node_type *) msg(command_class,"new")

Instance Variables:

command_num : command id.

Instance Methods:

setCommand
funciton: Defines the command box.
msg(command,"setCommand",[command text string],[command id])

getCommand
function: Answers with the command id.
id = (int) msg(command,"getCommand")

moveTo
function: Moves (centers) a command over a specific point.
msg(command,"moveTo",[x], [y])

draw
function: Draws a command box.
msg(command,"draw")

internals
function: Prints the command objects' instance variables.
msg(command,"internals")

Class: cursor

Super Class: object

Class Variables: none

Class Methods:

new
function: instantiates a cursor object.
cursor = (node_type *) msg(cursor_class,"new")

Instance Variable Names:

x,y	:	location of cursor
color	:	color of cursor bounding box
status	:	status of cursor ON or OFF
left,bottom	:	lower left hand corner of cursor bounding box
right,top	:	upper right hand corner of cursor bounding box

Instance Methods:

setColor
function: sets the color of the cursor's bounding box.
msg(cursor,"setColor", [COLOR])

setCursor
function: define the cursors' bounding box.
msg(cursor,"setCursor", [an_object])

updateCursor
function: updates the cursors' position and apperance.
msg(cursor,"updateCursor")

turnOn
function: turns the cursor on. Display the bounding box.
msg(cursor,"turnOn")

turnOff

function: turns the cursor off. No bounding box is displayed.
msg(cursor,"turnOff")

internals

function: prints the instance variables of the cursor object.
msg(cursor,"internals")

Class: dialogue

Super Class: window
text list
graphic list

Class Variables: none

Class Methods:

new
function: Instantiates a dialogue object.
dialogue = (node_type *) msg(dialogue_class,"new")

Instance Variables: none

Instance Methods:

appendText
function: Adds textual information to the dialogue.
msg(dialogue,"appendText", [text string], [font size])

appendCommand
function: Adds command boxes to the dialogue.
msg(dialogue,"appendCommand",[command text],[command id],[x],[y])

moveTo
function: Moves (centers) the dialogue object over a specific point.
msg(dialogue,"moveTo",[x], [y])

draw
function: Draws the dialogue box.
msg(dialogue,"draw")

engageInDialogue
function: Activates a dialogue box.
msg(dialogue,"engageInDialogue")

internals

function: Prints the dialogues' instance variables.

msg(dialogue,"internals")

Class: display window

Super Class: window
graphic list

Class Variables: none

Class Methods:

new
function: instantiates a display window object.
dis_window = (node_type *) msg(display_window_class,"new")

Instance Variables: none

Instance Methods:

update
function: Draws all the objects contained in the window.
msg(dis_window,"update")

internals
function: Prints the display window's instance variables.
msg(dis_window,"internals")

Class: mouse

Super Class: object

Class Variables:

x,y : location of mouse.
button : active buttonon mouse.

Class Methods:

track
function: Answers with the mouses' current location.
reply = (mouse_event_type *) msg(mouse_class,"track")

waitForEvent
function: Waits until a button is pushed. Answers with the mouses' location and button number.
reply = (mouse_event_type *) msg(mouse_class,"waitForEvent")

getMouse
function: Answers with the mouse's current location and button number if a button has been pushed, else button number is returned as zero.
reply = (mouse_event_type *) msg(mouse_class,"getMouse")

clearMouse
function: Resets the mouse and clears any queued events.
msg(mouse_class,"clearMouse")

Instance Variables: none

Instance Methods: none

Class: option

Super Class: window

Class Variables: none

Class Methods:

new
function: Instantiates an option object.
option = (node_type *) msg(option_class,"new")

Instance Variables: none

Instance Methods:

hiLite
function: Changes the appearence of an option.
msg(option,"hiLite", [ON or OFF])

setOption
function: Associates an object with the option.
msg(option,"setOption",[object])

getOption
function: Answers with a handle to the associated object.
object_handle = (node_type *) msg(option,"getOption")

draw
function: Draws the option.
msg(option,"draw")

internals
function: Prints the options instance variables.
msg(option,"internals")

Class: option window

Super Class: window
graphic list

Class Variables: none

Class Methods:

new
function: Instantiates a option window object.
opt_window = (node_type *) msg(option_window_class,"new")

Instance Variables:

left,bottom	:	bottom left hand corner of option window.
right, top	:	top right hand corner of option window.
delta_x	:	displacement of the window along the x-axis.
delta_y	:	displacement of the window along the y-axis.

Instance Methods:

defineOptionWindow
function: Defines the size of each individual window.
msg(opt_window,"defineOptionWindow",[left],[bottom],[right],[top])

addTo
function: Adds an option to the option window.
msg(opt_window,"addTo",[option])

moveTo
function: Moves (centers) the first option over a specific point.
All other options in the window are relative to this point.
msg(opt_window,"moveTo",[x],[y])

clear
function: Resets the option window.
msg(opt_window,"clear")

update

function: Answers with the selected option. The appearence of the opt window is modified to reflect the selection.

```
selection = (node_type *) msg( opt_window,"update",[x],[y] )
```

draw

function: Draws the option window.

```
msg( opt_window,"draw" )
```

internals

function: Prints the option window's instance variables.

```
msg( opt_window,"internals" )
```

Class: pull down menu

Super Class: window
graphic list

Class Variables: none

Class Methods:

new
function: Instantiates a pull doen menu object.
menu = (node_type *) msg(pull_down_menu_class,"new")

Instance Variables: none

Instance Methods:

appendMenuItem
function: Adds a menu item to the menu.
msg(menu,"appendMenuItem",[item text], [item id])

moveTo
function: Defines (centers) where first menu item is drawn.
msg(menu,"moveTo",[x],[y])

draw
function: Draws the pull down menu.
msg(menu,"draw")

engageInDialogue
function: Maintains the cursor within the menu.
msg(menu,"engageInDialogue")

internals
function: Prints the menus' instance variables.
msg(menu,"internals")

Class: window

Super Class: rectangle

Class Variables: none

Class Methods:

new
function: Instantiates a window object.
window = (node_type *) msg(window_class,"new")

Instance Variables:

backgroundcolor : color of the windows' background

Instance Methods:

setBackgroundColor
function: Sets the windows' background color.
msg(window,"setBackgroundColor",[color])

getBackgroundColor
function: Answers with the windows' background color.
color = (int) msg(window,"getBackgroundColor")

setWindow
function: Defines the window size.
msg(window,"setWindow", [left],[bottom],[right],[top])

clear
function: Clears the window with the windows' background color.
msg(window,"clear")

erase
funciton: Clears the windows contents and frame with background color.
msg(window,"erase")

clip

function: Bounds all drawings inside the window.
msg(window,"clip")

internals

function: Prints the windows' instance variables.
msg(window,"internals")

Class: graphics

Super Class: object

Class Variables: none

Class Methods:

startUp
function: Initializes the graphics environment.
msg(graphics_class,"startUp")

shutDown
function: Exits from the graphics environment.
msg(graphics_class,"shutDown")

setScreenMode
function: Sets the way images are drawn on the screen
(i.e. Normal, XOR, AND).
msg(graphics_class,"setScreenMode",[mode])

setOutputMode
funciton: Sets the mode in which output is sent to the display
processeor (i.e. graphics or alphanumeric).
msg(graphics_class,"setOutputMode",[mode])

drawPoint
function: Draws a point in the graphics environment.
msg(graphics_class,"drawPoint",[x],[y],[color])

drawRect
function: Draws a rectangle in the graphics environment.
msg(graphics_class,"drawRect",[left],[bottom],[right],[top],[color])

drawCircle
function: Draws a circle in the graphics environment.
msg(graphics_class,"drawCircle",[x],[y],[color])

drawPolygon
function: Draws a polygon in the graphics environment.
msg(graphics_class,"drawPolygon",[poly_object])

drawText
function: Write a text string in the graphics environment.
msg(graphics_class,"drawText",[x],[y],[size],[color])

solidFill
function: Enables or disables solid filling of shapes.
msg(graphics_class,"solidFill",[flag])

flushEvents
function: Removes all events from the event queue.
msg(graphics_class,"flushEvent")

showCrossHairs
function: Display the cursor crosshairs.
msg(graphics_class,"showCrossHairs")

hideCrossHairs
function: Hides the cursor crosshairs.
msg(graphics_class,"hideCrossHairs")

Instance Variables: none

Instance Methods: none

Class: graphic list

Super Class: link list

Class Variables: none

Class Methods:

new
function: Instantiates a graphic list object.
g_list = (node_type *) msg(graphic_list_class,"new")

Instance Variables: none

Instance Methods:

whoOwnsPoint
function: Answers with the object that contains the point.
owner = (node_type *) msg(g_list,"whoOwnsPoint",[x],[y])

insertByArea
function: Adds an object to the list according to the object's area (acsending order).
msg(g_list,"insertByArea",[object])

internals
function: Prints the instance variables of all the objects in the list.
msg(g_list,"internals")

Class: link list

Super Class: object

Class Variables: none

Class Methods:

new
function: instantiates a link list object.
list = (node_type *) msg(list_class,"new")

Instance Variables:

member_count : number of elements in the list.
head : first element in the list.
tail : last element in the list.

Instance Methods:

isEmpty
function: Answer whether the list contains any elements.
reply = (BOOLEAN) msg(list,"isEmpty")

getCount
function: Answers with the number of elements in the list.
count = (int) msg(list,"getCount")

addTo
function: Appends an object to the list.
msg(list,"addTo", [an_object])

AddToFront
function: Adds an object to the front of the list.
msg(list,"addToFront",[an_object])

insertBefore
function: Inserts an object into the list before a specific location.
msg(list,"insertBefore", [location], [an_object])

getObject
function: Answers with the n^{th} object from the list.
object = (node_type *) msg(list,"getObject", [n])

deleteObject
function: Removes an object from the list.
msg(list,"deleteObject", [an_object])

freeObjects
funciton: Deletes all objects from the list.
msg(list,"freeObjects")

Class: text list

Super Class: link list

Class Variables: none

Class Methods:

new
function: Instantiates a text list object.
t_list = (node_type *) msg(text_list_class,"new")

Instance Variables: none

Instance Methods:

startWriteAt
function: Defines the starting location to begin writing the first text object in the list. The remaining text object will begin at the same x cordinate but will be offset in the y direction.
msg(t_list,"startWriteAt",[x],[y])

internals
function: Prints the instance variables of all the text objects in the list.
msg(t_list,"internals")

Appendix E: Hardware Initialization

The Raster Technologies Model One/25 graphic processor can be configured to communicate with a number of different host computers. It is essential that a proper configuration be established with the host computer to ensure meaningful communication.

The VAX 11/785 requires that all data sent between it and the Model One/25 be formated as 7 bits, even parity. It is critical that the Model One/25 be configured for 7 bits, even parity communication before any attempt is made to run the ALC protoype software. If not configured properly, all data sent to the Model One/25 will be ignored. The following steps outline a procedure for configuring the Model One/25 to handle 7 bit, even parity data.

Step 1: 'Cold Boot' the Model One/25.

This is performed by pressing the 'cold boot' button located on the right-hand rear corner of the Model One/25 processor. This action will cause the Model One/25 to boot with its current configuration.

Step 2: Enter 'graphics mode'.

From the alphanumeric terminal connected to the Model One/25 enter a <CTRL D> or <CTRL E>. The system should respond with a exclamation point (!). The exclamation point indicates that the Model One/25 is in 'graphics mode'.

TABLE VIII
Communication Configuration for the Model One/25 and the VAX 11/785

PORT	RTS	CTS	STOP BITS	HIN	HOUT	CTRL	PARITY	BAUD
ALPHASIO	OFF	OFF	2	8	ON	OFF	ON	NONE	9600
MODEMSIO	OFF	OFF	1	8	ON	OFF	OFF	NONE	1200
GRINSIO	OFF	OFF	2	7	OFF	OFF	OFF	NONE	1200
TABLETSIO	OFF	OFF	2	8	OFF	OFF	OFF	NONE	1200
KEYBSIO	OFF	OFF	1	8	ON	OFF	ON	NONE	300
HOSTSIO	OFF	OFF	2	7	OFF	ON	ON	EVEN	9600

IEEE port : mode = off address = 0000

Host mode is HEXASCII

ROM sequence number is 001

Special Characters :

EntGr	Break	Warm	Kill	BS	ACK	Abort	Debug	HON	HOFF
0005	0010	001B	0040	0008	0007	0015	0018	0011	0013

Step 3: Display the current configuration.

Type discfg at the prompt.

! discfg <CR>

The current Model One/25 configuration should be displayed. Check this configuration against the configuration shown in Table VIII. If the HOSTSIO, Host Mode, and Special characters are the same, then the Model One/25 is correctly configured. Skip the remaining steps in this procedure by typing:

! quit <CR>

otherwise, proceed with the following steps.

Step 4: Set Special Characters.

If the special characters are correct proceed to step 5, otherwise enter the following at the prompt:

I spchar 0,1,5 <CR>

I spchar 5,1,7 <CR>

This will set the 'enter graphics' character to a <CTRL E> and the 'acknowledge' character to a <CTRL G>.

Step 5: Set HOSTSIO Line.

If the HOSTSIO line is correct proceed to step 6, otherwise enter the following at the prompt:

I syscfg serial hostsio rts off cts off stop 2 bits 7 parity e baud 9600 xin off xout on ctrl on <CR>

The system should respond with:

are you sure?

Answer:

yes <CR>

The Model One/25 will perform a 'warm boot', exiting you from the 'graphics mode'. Reenter 'graphics mode' by typing <CTRL E>. At the prompt type discfg and verify that the HOSTSIO line that was entered is correct. If it is incorrect, repeat this step, otherwise continue.

Step 6: Set Host Mode.

If the host mode is correct (i.e. HEXASCII), proceed to step 7, otherwise set the host mode by typing:

I syscfg host hostsio ascii <CR>

The system should respond with:

are you sure?

Answer:

yes <CR>

Again the Model One/25 will perform a 'warm boot'. Reenter 'graphics mode' with a <CTRL E>. Type discfg, verify the results, and repeat this step if necessary.

Step 7: Saving the Configuration.

To save this configuration to the Model One/25s' non-volatile memory, type:

! savcfg <CR>

The system should respond with:

are you sure?

Answer:

yes <CR>

At this point the correct configuration has been saved. Communication between the VAX 11/785 and the Model One/25 graphic processor is now possible.

Since all configuration data is stored in non-volatile memory, this configuration should remain until the configuration is physically modified again. Performing a 'cold boot' or powering down the Model One/25 will not change this configuration.

WARNING, WARNING, WARNING:

Configuration changes should not be attempted while the Model One/25 is connected (logged on) to the VAX 11/785. Configuration changes at this

time will usually cause the Model One/25 to lock up. There is no definite way of 'unlock' the Model One/25. Sometimes the system will remain idle for hours before it will respond.

Appendix E: Device Drivers

The functions implemented in this device driver package represent just a small set of available functions needed to fully exploit the capabilities of the Raster Technologies Model One/25 graphic system.

All modules implemented in this package were written in 'C'. They perform no error checking on operand values. They are, however, syntactically identical to graphic routines defined in the Raster Technologies Programming Guide [Raster Technologies, 1983].

The internal processing of the modules were modeled after an earlier Pascal implementation [Suzuki, 1983]. Basically, they convert an interger value (operator or operand) into a hexidecimal ascii string representation. This string is then sent to the Model One/25 via the 'putchar' command. The model One/25 interprets the string and performs the desired operation. For example, the opcode to clear the display screen is 135. This value is converted to a ascii string of '87' and sent to the Model One/25, interpreted and the screen cleared. All modules defined in this package function similarly.

A listing of the implemented modules follows, accompanied by a brief explanation. The reader should refer to the Raster Technologies Programming Guide for complete description of these modules and other commands supported by the Model One/25.

ack() : sends an acknowledgment (octal 07) to the Model One/25 after an read.

alpha_mode() : puts the Model One/25 into an alpha_numerics mode.

buttbl(i,m) : assigns a macro, m, with a particular mouse button,i.

circle(r) : draws a circle of radius r at the current point.

clear() : clears the display screen with the current color.

cload(r,x,y) : loads coordinate register, r, with x,y.

cmove(d,s) : copies the contents of coordinate register, s, into coordinate register d.

cororg(x,y) : sets the coordinate origin register to x,y.

drwabs(x,y) : draws a line from the current point to x,y.

flush() : empties the event queue.

graph_mode() : puts the Model One/25 into graphics mode.

macdef(n) : begins the n^{th} macro definition.

macend() : ends a macro definition.

macera(n) : clears the n^{th} macro definition.

moddis(flag) : changes the displays address mode.
flag = 0 : 512 x 512
= 1 : 1024 x 1024

movabs(x,y) : changes current point to x,y.

pixfun(mode) : sets the way in which images are drawn on the screen.
mode = 0 : normal
= 4 : XOR
= 5 : OR
= 6 : AND

point() : displays the current point.

prmfil(flag) : sets the primitive to be filled or unfilled.
flag = 0 : unfilled
= 1 : filled

readbu(flag,ctlr,button,x,y) : returns the function button and cursor location.

readcr(r,x,y) : returns the values x,y from the register r.

rectan(x,y) : draws a rectangle with the lower left hand corner at the current point and the upper right hand corner a x,y.

scrorg(x,y) : sets the screen coordinate register to x,y.

textl(string) : draws a text string starting at the current point.

textc(s,a) : specifies size (s) and angle (a) of next text draw.

value(r,g,b) : changes the current pixel color.
0 <= r,g,b <= 255

window(x1,y1,x2,y2) : defines the clipping window.

xhair(n,flag) : enables crosshair n.
flag = 0 : disable
= 1 : enable

Bibliography

Abbott, R.J. "Program Design by Informal English Descriptions," Communications of the ACM, 26: 882-894 (November 1983).

Abi-Ezzi, Salim S. and Albert J. Bunshaft. "An Implementer's View on PHIGS," IEEE Computer Graphics and Applications, 6: 12-23 (February 1986).

Adams, Karyl. A Display Environment Supporting the Interactive Generation of alphaNumerics and Symbology with DESIGNS on the Future. MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, May 1985.

-----. Instructor, Personal Interview, Air Force Institute of Technology, School of Engineering, Wright-Patterson AFB OH, April through May 1986.

Aho, Alfred V. and Jeffery D. Ullman. Principles of Compiler Design, Reading: Addison-Wesley Publishing Company, 1979.

Air Force Wright Aeronautical Laboratory, Advanced Cockpit/Crew Station Research Laboratory Technical Program Plan, August 1985.

Apple Computer Inc. Inside Macintosh: Promotional Edition, March 1985.

Arora, Adarsh K. and others. "An Overview of the VISE Visual Software Development Environment," IEEE Proceedings on Computer Software and Applications, 9th International Conference, 464-471, IEEE Press, New York, 1985.

Bertino, E. "Design Issues in Interactive User Interfaces," Interfaces in Computing, 3: 37-53 (February 1985).

Booch, Grady. "Object-Oriented Development," IEEE Transaction on Software Engineering, SE-12: 211-221 (February 1986).

-----. Software Engineering with Ada, Menlo Park: The Benjamin/Cummings Publishing Company. 1983.

Borgida, A., S. Greenspan and J. Mylopoulos. "Knowledge Representation as the basis for Requirements Specification," IEEE Computer, 18: 82-91 (April 1985).

Braaten, Capt. Alan J. and Capt. Markoe S. Hanson. An Object-Oriented Implementation Using Discriminate Record Types, Working Paper, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, October 1986.

Clemons, Eric K. and Arnold J. Greenfield. "The SAGE System Architecture: A System for the Rapid Development of Graphics Interfaces for Decision Support," IEEE Computer Graphics and Applications, 5: 38-50 (November 1985).

Constantine, L. L. and E. Yourdan. Structured Design, Englewood Cliffs: Prentice-Hall. 1979.

Cox, Brad J. "Message/Object Programming: An Evolutionary Change in Programming Technology," IEEE Software,1: 50-61 (January 1984).

-----. "The Object Oriented Pre-Compiler," SIGPLAN Notices,18: 15-22 (January 1983).

Curry, Gael A. and Robert M. Ayers. "Experience with Traits in the Xerox Star Workstation," IEEE Transaction on Software Engineering,SE-10: 519-527 (September 1984).

Denny, Capt. Michael W. The Graphical Kernel System: A Standards Implementation, MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

EVB Software Engineering Inc. An Object Oriented Design Handbook for Ada Software, 1985.

Fairley, R. E. Software Engineering Concepts, New York: McGraw-Hill, Inc., 1985.

Foley, James D. and Charles F. McMath. "Dynamic Process Visualization," IEEE Computer Graphics and Computer Applications,6: 16-25 (March 1986).

Foley, James D. and Andries Van Dam. Fundamentals of Interactive Computer Graphics. Reading: Addison-Wesley Publishing Company, 1982.

Glinert, Enhraim P. and Steven L. Tanimoto. "Pict: An Interactive Graphical Programming Environment," IEEE Computer, 17: 7-25 (November 1984).

Goldberg, Adele and David Robson. Smalltalk-80 The Language and its Implementation. Reading: Addison-Wesley Publishing Company, 1983.

Goldberg, Adele and Joan Ross. "Is the Smalltalk-80 System for Children?" Byte,6: 348-368* (August 1981).

Hanson, Capt. Markoe S. An Application of Advanced Ada Language Features to Data Structures in a Graphics Programming Environment. MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

Harslem, Eric. "A New Wave User Interface for the Office and the Airplane?" IEEE Spring COMPCON'84. 104-106. IEEE Computer Press, Los Alamitos, 1984.

Hearn, Donald and Pauline Baker. Computer Graphics. Englewood Cliffs: Prentice-Hall, 1986.

Hollan, James D. and others. "Steamer: An Interactive Inspectable Simulation-Based Training System," The AI Magazine,5: 15-27 (Summer1984).

Jackson, M. System Development. Englewood Cliffs: Prentice-Hall 1983.

Kramer, Axel. "IconMaker: Interactive User Interface Design", Workshop on Visual Languages. 192-198. IEEE Press, Silver Springs, 1984.

Lubinski, Th. and I. Hutzel. "An Object-Oriented Graphical Kernel System," Computer Graphics World: 70-75 (July 1984).

Marcus, Aaron. "Corporate Identity for Iconic Interface Design: The Graphic Design Perspective," IEEE Computer Graphics and Applications, 4: 24-36 (December 1984).

Newman, William M. and Robert F. Sproull. Principles of Interactive Computer Graphics (Second Editon). New York: McGraw-Hill Book Company, 1979.

Ohlson, Mark. "System Design Guidelines for Graphics Input Devices," Tutorial: Computer Graphics, New York: IEEE Computer Society, 1979.

Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, 15: 1053-1058 (December 1972).

Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," Byte, 11: 134-144 (August 1986).

Raeder, Georg. "A Survey of Current Graphical Programming Techniques," IEEE Computer, 18: 11-25 (August 1985).

Raster Technologies. Raster Technologies Model One/25 Programming Guide, Revision 4.1. Decemeber 1983.

Reiss, Steven P. "An Object-Oriented Framework for Graphical Programming," SIGPLAN Notices, 21: 49-57 (October 1986).

Rentsch, Tim. "Object Oriented Programming," SIGPLAN Notices, 17: 51-57 (September 1982).

Robson, David. "Object-Oriented Software Systems," Byte, 6: 74-86+ (August 1981).

Rose, Capt. Kevin W. Development of an Interactive Computer Graphics System Library and Graphic Tools. MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982.

Rubinstein, Richard and Harry Hersh. The Human Factor: Designing Computer Systems for People, Digital Press: The Digital Equipment Corporation. 1984.

Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages," IEEE Computer, 16: 57-69 (August 1983).

Singh, Baldev and others. "A Graphics Editor for Benesh Movement Notation," Computer Graphics, 17: 51-62 (July 1983).

Shooman, Martin L. Software Engineering. New York: McGraw-Hill, Inc., 1983.

Smith, David Canfield and others. "Designing the Star User Interface," Byte7: 242-282+ (April 1982).

Sommerville, Ian. Software Engineering (Second Edition). Reading: Addison-Wesley Publishing Company, 1985.

Stefik, Mark and Daniel G. Bobrow. "Object-Oriented Programming: Themes and Variation," The AI Magazine,6 : 40-62 (Winter 1986).

Stroustrup, Bjarne. "Classes: An Abstract Data Type Facility for the C Language," SIGPLAN Notices,17: 42-51 (January 1982).

Sutherland, Ivan E. "Sketchpad A Man-Machine Graphical Communication System," (orig. Published 1963), Tutorial and Selected Readings in Interactive Computer Graphics, edited by Herbert Freeman. 2-19. Silver Springs: IEEE Computer Society Press, 1980.

Suzuki, Second Lieutenant Laura R.C. The Addition of Advanced Scene Rendering Techniques to a General Purpose Graphics Package. MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH,December 1984.

Warnier, M. Logical Construction of Programs. New York: Van Nostrand, 1977.

Wirth, N. "Program Development by Stepwise Refinement," Communications of the ACM, 14: 221-227 (April 1971).

Wisskirchen, Peter. "Towards Object-Oriented Graphics Standards," Computers and Graphics,10: 183-187 (1986).

Vita

Captain Alan J. Braaten was born on 28 October 1957 in Madison, Wisconsin. He graduated from high school in Langdon, North Dakota, in 1975. He attended Colorado State University from which he received the degree of Bachelor of Science in Computer Science in 1980. Upon graduation, he received a commission in the USAF through the ROTC program. He was employed as systems analyst at the National Military Command Center, Washington, D.C., until entering the School of Engineering, Air Force Institute of Technology, in June 1985.

Permanent address: 3255 South Dexter,
Denver, Colorado 80222

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A178-636

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/MA/86D-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering	6b. OFFICE SYMBOL (If applicable): AFIT/ENC
6c. ADDRESS (City, State, and ZIP Code): AirForce Institute of Technology, Wright-Patterson AFB, Ohio 45433-6583	7a. NAME OF MONITORING ORGANIZATION
	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: FLIGHT DYNAMICS LABORATORY	8b. OFFICE SYMBOL (If applicable): FIGR
8c. ADDRESS (City, State, and ZIP Code): Air Force Wright Aeronautical Laboratories, Flight Dynamics Laboratory (FIGR), Wright-Patterson AFB, OH 45433-6583	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): A GRAPHICS ENVIRONMENT SUPPORTING THE RAPID PROTOTYPING OF PICTORIAL COCKPIT DISPLAYS

12. PERSONAL AUTHOR(S): Alan J. Braaten, B.S., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1986 December	166

16. SUPPLEMENTARY NOTATION

17. COSATI CODES: FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): INTERACTIVE GRAPHICS; COCKPIT DISPLAYS

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Karyl Adams
Instructor of Computer Science

Approved for public release: IAW AFR 190-1.
5 March 87
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Karyl Adams, Instructor	255-3098	AFIT/ENC

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

The purpose of this investigation was to design and implement a graphics based environment capable of supporting the rapid prototyping of pictorial cockpit displays. Attention was focused on the interactive construction of pictorial type cockpit displays from libraries of cockpit displays and symbology.

Implementation was based on an object-oriented programming paradigm. This approach provided a natural and consistent means of mapping abstract design specifications into functional software. Implementation was supported by an object-oriented extension to the 'C' programming language.

Although this investigation addressed a specific application, the resulting graphic environment is applicable to other areas requiring the rapid prototyping of pictorial displays.